

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

**IMPLEMENTATION OF A MULTI-AGENT SIMULATION
FOR THE NPSNET-V VIRTUAL ENVIRONMENT
RESEARCH PROJECT**

by

David B Washington

September 2001

Thesis Advisor:

Second Reader:

Michael Capps

Don McGregor

Approved for public release; distribution is unlimited.

Report Documentation Page

Report Date 30 Sep 2001	Report Type N/A	Dates Covered (from... to) -
Title and Subtitle Implementation of a Multi-Agent Simulation for the NPSNET-V Virtual Environment Research Project	Contract Number	
	Grant Number	
	Program Element Number	
Author(s) David B. Washington	Project Number	
	Task Number	
	Work Unit Number	
Performing Organization Name(s) and Address(es) Research Office Naval Postgraduate School Monterey Ca. 93943-5138	Performing Organization Report Number	
Sponsoring/Monitoring Agency Name(s) and Address(es)	Sponsor/Monitor's Acronym(s)	
	Sponsor/Monitor's Report Number(s)	
Distribution/Availability Statement Approved for public release, distribution unlimited		
Supplementary Notes		
Abstract		
Subject Terms		
Report Classification unclassified	Classification of this page unclassified	
Classification of Abstract unclassified	Limitation of Abstract UU	
Number of Pages 117		

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2001	3. REPORT TYPE AND DATES COVERED Masters Thesis	
4. TITLE AND SUBTITLE: Title (Mix case letters) Implementation of a Multi-Agent Simulation for the NPSNET-V Virtual Environment Research Project			5. FUNDING NUMBERS	
6. AUTHOR(S): David B. Washington				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Distribution Statement (mix case letters)			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>Traditional networked military simulation systems are technologically frozen the moment they are completed, thus limiting the participants that can interact in the simulation. When training for urgent missions in emerging conflict areas, assimilating new models, threat behaviors, and new terrain environments into the simulators requires lengthy integration, is prohibitively costly, and is non-distributable electronically at runtime. Threat behaviors are pre-scripted, lack organization, and do not accurately portray doctrine or rules-of-engagement.</p> <p>NPSNET-V is a novel architecture for networked simulations that supports scalable virtual worlds with built-in dynamic entity loading. These advances address each of the above concerns: scalability, entity and environment distribution, and dynamic technology loading. By combining this architecture with a system for creating autonomous, adaptable agents, threat forces can be accurately simulated. This architecture is useful for proposing designs for strategies, tactics, or force packages during the conduct of experiments.</p> <p>The result of this thesis is a proof-of-concept application demonstrating the utility of these architectural advances. In this application, numerous autonomous agents form complex, dynamic, and adaptable interactions with resident and remote heterogeneous entities. These results define the course for future military models and simulations.</p>				
14. SUBJECT TERMS Multi-Agent Simulation, agent-based simulation, adaptive agents, autonomous agents, networked virtual environment.			15. NUMBER OF PAGES 101	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**IMPLEMENTATION OF A MULTI-AGENT SIMULATION FOR THE NPSNET-
V VIRTUAL ENVIRONMENT RESEARCH PROJECT**

David B. Washington
Major, United States Army
B.S., Tulane University, 1990

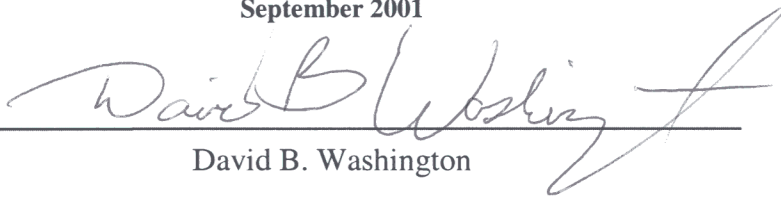
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

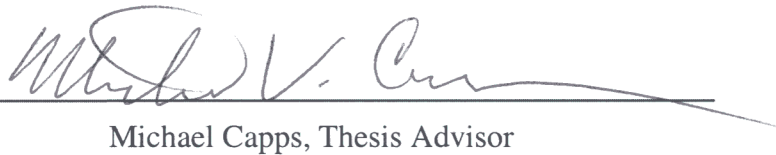
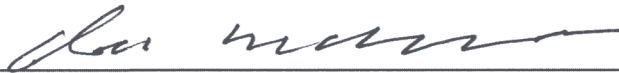
from the

**NAVAL POSTGRADUATE SCHOOL
September 2001**

Author:


David B. Washington

Approved by:


Michael Capps, Thesis Advisor

Don McGregor, Second Reader



Chris Eagle, Chairman
Computer Science Department

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Traditional networked military simulation systems are technologically frozen the moment they are completed, thus limiting the participants that can interact in the simulation. When training for urgent missions in emerging conflict areas, assimilating new models, threat behaviors, and new terrain environments into the simulators requires lengthy integration, is prohibitively costly, and is non-distributable electronically at runtime. Threat behaviors are pre-scripted, lack organization, and do not accurately portray doctrine or rules-of-engagement.

NPSNET-V is a novel architecture for networked simulations that supports scalable virtual worlds with built-in dynamic entity loading. These advances address each of the above concerns: scalability, entity and environment distribution, and dynamic technology loading. By combining this architecture with a system for creating autonomous, adaptable agents, threat forces can be accurately simulated. This architecture is useful for proposing designs for strategies, tactics, or force packages during the conduct of experiments.

The result of this thesis is a proof-of-concept application demonstrating the utility of these architectural advances. In this application, numerous autonomous agents form complex, dynamic, and adaptable interactions with resident and remote heterogeneous entities. These results define the course for future military models and simulations.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	THESIS STATEMENT	1
B.	MOTIVATION	1
1.	Monterey Bay Aquarium	2
2.	Autonomous, Adaptable Agents	3
a.	Interaction with New, Unknown Agents	3
b.	Learning	4
c.	Master/Ghost	5
3.	Dynamic Behaviors.....	6
4.	Networked Virtual Environment	7
C.	APPROACH	7
1.	Integrate RELATE with NPSNET-V Model, View, Controller	7
a.	Create New Agent Types	9
b.	Add New Agent Behaviors	9
c.	Agents' Distinct Personalities	10
d.	Genetic Algorithm Replicates Natural Selection.....	11
2.	Creating New NPSNET-V Applications	12
3.	Proof of Concept: <i>FishWorld</i>	13
D.	PROBLEM.....	14
1.	How to Give RELATE Behaviors to NPSNET-V Entities.....	14
2.	How to Extend the Features of NPSNET V.....	14
a.	How to Interface with Dynamically Added Agents.....	14
b.	How to Solve Collision Detection with a Complex Environment	15
c.	How to Solve Convergence and Dead Reckoning for Ghosts	15
3.	How to Create a Realistic Underwater Environment.....	15
a.	How to Solve Physics of Fish Including Aquarium Wave Motion ..	16
b.	How to Create Realistic Kelp	16
c.	How to Create Realistic Underwater Environment	16
E.	THESIS ORGANIZATION.....	17
II.	BACKGROUND	19
A.	TECHNOLOGIES	19
1.	Multi-Agent Simulation.....	20
2.	RELATE	20
3.	Genetic Algorithms	21
4.	Java3D.....	23
5.	VRML.....	23
B.	RELATED WORK	24

1.	NPSNET-V	24
a.	Lightweight Directory Access Protocol (LDAP).....	24
b.	Entity Dispatcher.....	26
c.	Area of Interest Manager	27
d.	Dynamic Protocol/Entity Discovery.....	28
e.	Dynamic Network Optimization.....	28
2.	Kelp Forest Modeling Project.....	29
a.	Static Path Animation	29
b.	Static Environment.....	29
3.	Capture the Flag.....	30
a.	Distributed Interactive Simulation.....	30
b.	Multi-Agent Behaviors in Capture the Flag.....	31
4.	El Farol.....	31
a.	Adaptable Behaviors.....	32
b.	Dynamic Emergent Behavior.....	32
5.	Boids	33
a.	Flocking Behavior.....	33
b.	Terrain Avoidance.....	34
C.	CONCLUSION.....	35
III.	INTEGRATE RELATE WITH NPSNET-V MODEL, VIEW,	
CONTROLLER	37	
A.	RELATE IMPLEMENTATION	37
B.	NPSNET-V	39
1.	RELATE Agent to NPSNET-V EntityMaster	40
2.	NPSNET-V EntityGhost.....	43
3.	NPSNET-V View.....	46
C.	CONCLUSION.....	46
IV.	CREATING NEW NPSNET-V APPLICATIONS	49
A.	COLLISION DETECTION	50
1.	Entity-to-Environment Collisions	51
2.	Entity-to-Entity Collisions	51
B.	PHYSICS	53
C.	MODEL, VIEW, CONTROLLER	53
1.	Master.....	54
2.	Ghost	56
a.	Steering Commands	58
b.	Full State Protocol.....	59
c.	Fire Torpedo Command	59
d.	Suffer Attack	60
3.	View.....	61
4.	Controller.....	63
D.	CONCLUSION.....	63

V.	PROOF OF CONCEPT: <i>FISHWORLD</i>	65
A.	INTENT OF FISHWORLD.....	65
1.	Features	65
a.	Dynamic Heterogeneous Entity Discovery.....	65
b.	Scalability.....	66
c.	Application Generic Implementation.....	67
2.	Autonomous Entity Requirements	67
B.	INTERACTING WITH A DYNAMIC WORLD.....	68
1.	Learning	68
2.	Memory.....	69
3.	Genetic Algorithm.....	70
C.	CREATING NEW AGENT BEHAVIORS	71
1.	Roles.....	72
2.	Goals and Rules	72
a.	Dead Goal	73
b.	Avoid Wall Goal.....	73
c.	Keyboard Control Goal.....	75
d.	Avoid Collision Goal	76
e.	Flee Goal.....	77
f.	Eat Goal.....	77
g.	School Goal.....	79
h.	Cruise Goal	82
3.	Actions	82
D.	CREATING DISTINCT PERSONALITIES	83
1.	Aggressive.....	83
2.	Collision Wall	83
3.	Collision Fish.....	84
4.	Schooling Preferences.....	84
5.	Twistedness	85
6.	Hungriness	85
7.	Blindness	85
E.	CONCLUSION.....	85
VI.	CONCLUSION	87
A.	RESULTS	87
B.	CONCLUSION.....	88
C.	FUTURE WORK.....	89
1.	Agent Ghost Controller	89
2.	Dead Reckoning.....	90
3.	Agent Network Tuning	90
4.	Scalability Study.....	91
5.	Code-less Agent Creation	91
6.	Component Loading.....	92

7. Security	92
8. New Virtual Worlds.....	93
a. Warfighting Experiment (WE).....	93
b. Experiment with Tactics	94
c. Experiment with Systems	95
GLOSSARY.....	97
LIST OF REFERENCES	99
INITIAL DISTRIBUTION LIST	101

LIST OF FIGURES

Figure 1.	Model, View, Controller Design Paradigm.	8
Figure 2.	Genetic Algorithm.....	22
Figure 3.	NPSNET-V LDAP Directory Server.	25
Figure 4.	Entity Dispatcher Receive Sequence.....	27
Figure 5.	RELATE Decision Tree.....	38
Figure 6.	FishWorld EntityMasters	40
Figure 7.	EntityMaster Design.	41
Figure 8.	EntityGhost Design.	46
Figure 9.	Virtual World Layers.	50
Figure 10.	Entity-To-Entity Collision Detection.	51
Figure 11.	EntityMaster Layers.....	54
Figure 12.	EntityGhost Layers.....	57
Figure 13.	EntityView Layers.	61
Figure 14.	Local Max / Theoretical Max.....	70
Figure 15.	FishWorld Decision Tree.	71
Figure 16.	Aquarium Layout From Ref. [Brutzman, 2001].	75

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ABBREVIATIONS AND ACRONYMS

CAS	Complex Adaptive Systems
DIS	Distributed Interactive Simulation
DoD	Department of Defense
GA	Genetic Algorithm
GUID	Globally Unique Identifier
ISAAC	Irreducible Semi-Autonomous Adaptive Combat
LDAP	Lightweight Directory Access Protocol
MAS	Multi-Agent System
MOVES	Modeling, Virtual Environments and Simulation
MVC	Model, View, Controller
NPS	Naval Postgraduate School
PDU	Protocol Data Units
URL	Uniform Resource Locator
VE	Virtual Environment
VRDNS	Virtual Reality Domain Naming Service
VRML	Virtual Reality Modeling Language
WE	Warfighting Experiment

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THESIS STATEMENT

By combining a fully dynamic, scalable networked virtual environment (VE) with an interactive multi-agent simulation architecture, it is possible to develop virtual environments supporting a large number of dynamic, heterogeneous entities with complex, adaptable, and interactive behaviors. The dynamic VE architecture is useful in geographically distributed deployments where it is difficult to coordinate shutdown/restarts across global time zones. The multi-agent architecture is useful for developing evolving environments that do not have a completely specified list of entities or interactions. Environments developed by combining these architectures are powerful tools for solving complex problems, and can be configured at run-time without requiring a redesign of a structured, monolithic architecture. These problem spaces include those requiring rapid prototyping, detailed experimentation, or extensive testing/simulation. An aquarium setting will serve as a proof of concept environment, in which numerous autonomous agents interact with resident and remote heterogeneous entities to form dynamic, adaptable behaviors.

B. MOTIVATION

VEs with dynamic adaptable behaviors can help to solve a large class of problems that are complex and costly (including military training scenarios and simulations). NPSNET-V (Capps, 2000) provides a canvas for creating networked 3-D virtual world. Current networked games and simulations require prior knowledge of all entities that will

be used in the system, their graphical representations, implemented behaviors, and protocols. Adding new features requires shutting down, coding, integrating, testing, and recompiling the entire system. This is a shortcoming not inherent in the NPSNET-V architecture due to dynamic loading/entity-discovery capability. By using a plug-able Area of Interest Manager (AOIM), any NPSNET-V VE can be scaled to host a large number of participants. By combining these capabilities into a single project, the NPSNET-V architecture could likely advance work in military simulations, real-time prototyping, future-capabilities experiments, and on-line interactive games. Additionally, any user combining NPSNET-V with RELATE (an agent architecture discussed in Chapter II) can create agents that interact with others that have yet to be created without the use of deterministic methods.

1. Monterey Bay Aquarium

The NPSNET-V research group decided that the Monterey Bay Aquarium would be a useful first application. It is easily identifiable to the Monterey community and could bring positive attention to the research. This environment is densely populated and will fully test the true scalability limits of NPSNET-V (innovative techniques are used to conquer network bandwidth constraints—see Chapter II). In addition to density, the aquarium has a highly heterogeneous population of sea life. This population is a continuously dynamic population where new species are added and removed frequently. There exists a myriad of interactions between these continually changing characters. The result of this thesis is an application where autonomous agents will interact and adapt to

this dynamic, scalable environment. This creates unlimited possibilities for behavior modeling and testing. The application described in this thesis features the aquarium and is a proof-of-concept application that fully tests the capabilities of NPSNET-V. The name of this virtual world is *FishWorld*.

2. Autonomous, Adaptable Agents

The heterogeneous, autonomous agents that populate the application interact with others that are added dynamically. These agents learn about and adapt to the new additions without using deterministic algorithms in scripted behaviors. Not all the possible interactions are known before introduction. The determination of which agent will be dominant or most successful is left up to the agent that best adapts. If there are many different types of predators, the food chain is determined by natural selection. Rigid, non-adaptable agents may emerge dominant in the short term, but may in turn become dominated by agents that are adaptable. The ability to combine an agent architecture, like RELATE, and NPSNET-V will create a test-bed application useful for experimentation—integrating improvements from previous iterations for subsequent trials. The ability to iteratively test new subjects in an environment is the process for many industrial, scientific, and military experiments. This is especially useful for the simulation of human participants in automated forces.

a. Interaction with New, Unknown Agents

Before the U.S. Army committed to purchasing the Longbow Apache attack helicopter, the Aviation Battle Lab at Fort Rucker, AL had to conduct several

experiments to prove its worth and determine effective employment principles.

Designers had to create several dedicated software models to support these tests. These models were replicated in many software languages for dozens of different, preexisting applications. Many of the applications were reworked, because existing models required extensions to replicate the interactions with the Longbow. Because NPSNET-V allows for dynamically added entities, and because RELATE can give these entities autonomy and the capability for complex, adaptable interactions, the results of this thesis could be applied to applications used for experimentation such as those used for the Longbow. This exciting technology could be extended to an unlimited number of additional applications.

b. Learning

It would be valuable for military acquisitions of new systems if a simulation could replace how humans employ and test new weapon systems without requiring expensive human input for this tedious, arduous task. A simulation system can apply learning from previous iterations for employment in follow-on iterations. Only subjects that perform well or combinations of well performing subjects would be allowed to continue to subsequent testing. The computer could evaluate employment of the experimental weapon system in ways that a human may never have imagined. To accomplish this, agents must be created that learn. During the interactions such as those described above, a military system may be added to a virtual environment. It is unknown whether existing systems will continue to dominate in the VE. Adaptable, autonomous agents could have been used by the Aviation Battle Lab to suggest the most effective

employment principles of the Longbow in the confines of the simulated situation. By letting the simulation run, the agent could adapt to its environment, discover interactions with friends and foes, and learn about its own capabilities. After several iterations, an analysis could be conducted to determine how the agent had adapted and what the opposing force had done to counteract the agent's capabilities. This information could be used in a determination of the tested system's effectiveness.

c. Master/Ghost

In networked VEs, an agent will actually interact with the graphical representation of other agents remotely located elsewhere on the network. Entities update their position by transmitting packets, but this transmission can flood a network severely limiting the number of participants. Inaccuracies in packet transmission exist due to network latency, bandwidth restrictions, and limited buffer sizes further restricting the number of possible participants (Singhal and Zyda, 1999). Due to these network bottlenecks, the number of participants in a simulation of a battle is often limited to those of the smallest organizations—not those of realistic combat organizations—severely limiting the validity of military experiments and simulations. The requirement exists for a scalable, accurate interpretation of behaviors between agents and their graphical representations. The design paradigm for NPSNET-V labels the created entity a *Master*. The represented entity on a participating machine is labeled a *Ghost*. The *Master* responds to its environment and others as defined by its behavior. The *Ghost* mirrors the actions of the *Master* on remote, participating machines interested in the entity. A goal of this thesis is to give the *Ghost* enough intelligence that it can realistically emulate the

behavior of the *Master* while minimizing network updates from the *Master* to the *Ghost*. This must be balanced with the requirement for efficient use of CPU cycles. The representative *Ghost* should not monopolize the computational capacity of machines on which it resides which would violate the scalability requirements of NPSNET-V. Research in this area would provide a lightweight solution that could be extended to a myriad of military networked collective training simulations. See chapter VI for further discussion.

3. Dynamic Behaviors

Most current networked applications must have prior knowledge of all entities and their pre-scripted behaviors and animations. The protocols that call these events on the *Ghost* representatives must also be known in advance. With NPSNET-V, protocols, entities, and graphical representations can be created and added at runtime. All participating machines can access these components as that machine gains interest in the new item. If a new type of entity is created that has articulated limbs or fixtures, it can still be accurately displayed in *FishWorld*. No prior knowledge is required. This capability would have saved time, effort, and money in the above example of the Longbow experiments. The application would not have had to be extended and recompiled to include the capabilities of the Longbow. Adding new weapon types in current military simulations requires that the software be modified, integrated, recompiled and reloaded—which is often. This is needlessly expensive.

4. Networked Virtual Environment

Networked Virtual Environments are valuable tools for many tasks. The graphical representation of environments allows users to visualize the problem space they are interacting with. They are extremely useful for applications including design, training, experimentation, testing, and entertainment. In developing new products, interactions must be tested with other emerging technologies. The Longbow experiments, for example, could have been conducted simultaneously from several battle labs to test proper integration with other new developing systems. The technology of NPSNET-V and *FishWorld* would provide this type interaction without the need for extensive integration or the need to integrate heterogeneous applications into a monolithic, inextensible, homogeneous one.

C. APPROACH

This thesis will culminate with the creation of a Networked Virtual Environment hosting a multi-agent simulation of the Monterey Bay Aquarium Kelp Forest exhibit (see <http://www.mbayaq.org>). The following paragraphs of this section describe the approach that will be used.

1. Integrate RELATE with NPSNET-V Model, View, Controller

NPSNET-V uses the *Model*, *View*, and *Controller* design paradigm (Buschman, 1996). The model contains the state information of an entity (Figure 1 below). This model may include positional, orientation, and physical data such as velocities and

accelerations. The view is the graphical representation of the entity. The controller is what guides the behavior of the entity including computer control or user control through use of the keyboard, mouse, or game controller. An entity instance will have exactly one model. Several objects distributed throughout the entity object may compose this model, but an entity can only be uniquely identified by a single set of state information and a Globally Unique Identifier (GUID). Several entities may use the same model type, but the instance of one model will uniquely identify a single entity. This entity may have more than one view object over time—one for several, specific situations such as death, fleeing, or swimming; but it will only have one view at a time. Other entities may have unique instances of these same views as their graphical representation as well. The selection of the specific view will depend on the state of the owning entity’s model. This entity may have more than one controller as described above—keyboard controller, mouse controller, or agent behavior.

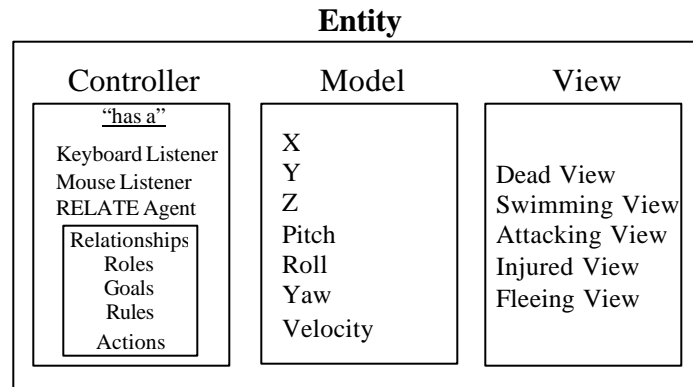


Figure 1. Model, View, Controller Design Paradigm.

For *FishWorld*, control is selectable. When under user control, the computer-controlled entities should continue to interact with the user-controlled entity. The

RELATE Java library is a multi-agent architecture that uses a goal-based reasoning decision-tree structure. The result of an agent's reasoning is a simple action such as turn, climb, or continue straight which provides autonomous control for agents. The challenge is to create a cohesive design that also allows for a user to take control of entities. This is a feature that provides vital extensibility for experimentation applications. In this type application, an agent could be driven to a required location by a user for repeatable scenarios to test new behavioral goal sets.

a. Create New Agent Types

To demonstrate the versatility of agent-based programming, a myriad of divergent agents will be created. Agent types can be completely heterogeneous for all applications that run on NPSNET-V. To demonstrate this capability both organic and man-made simulated objects will be created. Interactions can still occur in this amalgamated environment. This is important as a demonstration feature to prove NPSNET-V's use for military simulations or experiments. The type vehicles in military scenarios are quite varied. To create a dynamic environment for *FishWorld*, predators, prey, schooling agents, submarines, and surface ships all will be created—the more heterogeneous, the better.

b. Add New Agent Behaviors

Some of the agent behaviors will include schooling, searching, fleeing from predators, attacking, and others. The architecture of RELATE gives an agent that implements it a container of rules that it can use. A large set of rules will be constructed,

and varying which specific rules an agent has access to will create variations in the agents. This set of rules is not the limit that a new entity type can use. Entities can extend and modify these standard rules. This is important in a test of NPSNET-V's usefulness, because secondary and subsequent iterations of any system often add features and capabilities. These new capabilities could simply be modeled by the dynamic behavior capability of *FishWorld* and NPSNET-V.

c. Agents' Distinct Personalities

An agent hierarchy of dominance should not be predetermined. Dominance is something that agents should learn and adapt to. Additionally, this flexibility better suits the architecture of NPSNET-V. When new agents are added dynamically, this adaptability will allow agents to survive and learn about the newly created environment. To make interactions interesting, the individuals of a species have unique propensities for certain behaviors. These personality traits should have advantages, but should also have disadvantages. The desire to follow others in a school very closely should provide the protection afforded by being deep in a school, but possibly result in increased injury due to increased collisions with others. Increased aggressiveness should allow an agent to arise as a leader in a school, but this arrogance should make it more likely to be attacked. Personality traits in fish include aggressiveness, leadership, closeness, blindness, hungriness, and others.

The abilities of this technology are very powerful. During the description of the Longbow experiments in the previous chapter, employment principles were

analyzed. An autonomous Longbow agent could have been given a personality that described the adjustable features of the Longbow experiment. Many capabilities of the Longbow represent a trade-off. For example, if more weapons are carried, the aircraft can kill more, but has increased weight and decreased flight performance, and can be killed more easily. Several autonomous Longbow agents could be given distinct personalities and exercised in a simulation. The scores of the participating agents could represent the best solutions for Longbow development.

d. Genetic Algorithm Replicates Natural Selection

Many multi-agent simulations use an objective function to grade the actions of the participating agents. Based on the results of this objective function, an agent could change its active goal, rule, personality, or action. The agents in *FishWorld* can be harmed by collisions with other agents, collisions with the wall, and attacks from predators. When an agent dies, it is reintroduced with changes to its personality traits. The new design will be the result of a genetic algorithm run on the best two surviving agents of the same type—the parents for the new agent. This life and death struggle will act as the objective function. Using a genetic algorithm will be the basis for the adaptation. Introduction of mutations will ensure that new personalities can continually be discovered.

The use of a genetic algorithm would be a vital step to creating the next trial agent version in an ongoing experiment. The failure of an agent during an iteration of a multi-agent simulation is simply an opportunity to create the next generation agent.

This type failure mode in a military-type experiment is realistic—destruction of a vehicle can represent failure of a tactical strategy or vehicle capability. An objective function can also be used to grade the performance of an agent, because destruction may also occur on a successful entity. The more iterations of a simulation that are run, then the better the results of the experiment are likely to be. The objective function and genetic algorithm are highly tailored able for each application.

2. Creating New NPSNET-V Applications

After the agents are up and running, the second challenge is to implement interfaces that any newly created application can use to run on NPSNET-V. These interfaces must be fairly robust, because the possibilities for interactions between objects in the NPSNET-V virtual environment are completely unlimited. Newly created entities must be able to be added to any new virtual environment. These agents must be able to ask questions and gather information about the environment. They must be able to be affected by physical effects like wind or water currents. They must be able to ask questions of other agents within their sensing range in order to allow for interactions. These interactions need not be limited to only collision avoidance or collision detection. The creator of a new virtual environment on NPSNET-V gets to be the architect for these interactions in his world.

Creating a general interface that any NPSNET-V application can use to solve collision detection issues with the environment and physical interactions that act upon participating agents greatly increases the capability and extensibility of NPSNET-V. The

interface would allow agents to learn about their environment. Many experiments that extend the capabilities of this thesis could simply tailor the physics to match those of their experiment. This could prove useful for tests of a vehicle to land on Mars or the moon, tests of an amphibious vehicle that operates on land and sea, or test of a tilt-rotor vehicle that operates like a helicopter and an airplane.

3. Proof of Concept: *FishWorld*

FishWorld is a fully dynamic, scalable, networked application that creates a realistic, virtual underwater environment. It is a combination of this virtual environment with an interactive multi-agent simulation architecture, which supports a large number of dynamic, heterogeneous entities with complex, adaptable, and interactive behaviors.

FishWorld is the backdrop for interactions between a myriad of autonomous and user-controlled agents of varying types, each with unique personalities. This application is designed to test the capabilities of NPSNET-V. It is highly scalable, and will be able to host a large number of heterogeneous agents. New fish types will be able to interact with any other entity in the aquarium. The agents will be able to interact with the environment, be affected by currents, and be affected by environmental collisions.

FishWorld's agents will adapt and evolve as new interactions are created. No one entity type will be able to dominate the others in this virtual world as long as the architect of these entities builds in robust learning and adaptation.

This thesis will prove that an application can be constructed that allows for experimentation of dynamically loaded entities, dynamically discovered interactions, self-tuning behaviors and attributes, and entity adaptations and behavior modeling.

D. PROBLEM

There are several challenges that have been resolved for correct modification of NPSNET-V to allow for successful implementation of *FishWorld*.

1. How to Give RELATE Behaviors to NPSNET-V Entities

RELATE and NPSNET-V are quite separate architectures with different purposes. The main challenge is finding the best way to merge them into an extensible, cohesive package.

2. How to Extend the Features of NPSNET V

NPSNET-V entities can be created and added dynamically at run time. The components of these entities include the *Master*, *Ghost*, *View*, *Controllers*, and any unique protocols that define the entities' behaviors. Some problems areas to consider are:

a. How to Interface with Dynamically Added Agents

Hooks must be built to allow different *FishWorld* entities to interface with the resident autonomous agents that are instantiated at startup. These hooks must be robust enough to allow for emergent behavior, adaptation, and encourage learning.

b. How to Solve Collision Detection with a Complex Environment

While creating *FishWorld*, the interface that allows agents to learn about the environment should be extensible to numerous applications. In addition to solving collision detection inquiries, this interface should be useful for relaying physical properties of the environment such as water currents.

c. How to Solve Convergence and Dead Reckoning for Ghosts

In *FishWorld*, fish agents in a tight school may make several turns a second to stay in the school while avoiding collisions with others nearby. Sending packets at frame rate would violate the scalability goal of NPSNET-V. A solution must be devised that limits the number and size of packets sent per second. The dead reckoning schemes of the *Ghost* must be robust enough to interpret these packets and move realistically. The convergence algorithms must ensure smooth transitions toward corrected position updates. This behavior must meet the additional requirement of not monopolizing the CPU.

3. How to Create a Realistic Underwater Environment

Virtual environments should provide a realistic scene that causes the viewer to feel presence and immersion. The created scene must be captivating enough to encourage others to participate by creating entities of their own.

a. How to Solve Physics of Fish Including Aquarium Wave Motion

A decision by an autonomous agent must be turned into an actual step.

This decision may translate into an action to turn left or right or to continue straight. The agent may want to accelerate, decelerate, climb or descend. A goal-derived decision to flee from a predator, for example, must be translated into one of these actions.

Additionally, the currents of the Monterey Bay Aquarium and other physical forces must affect this action of Masters and Ghost.

b. How to Create Realistic Kelp

Swaying, flowing stalks of kelp dominate the Kelp Forest exhibit in the Monterey Bay Aquarium. The Kelp Forest Exhibit Modeling Project, <http://web.nps.navy.mil/~brutzman/kelp/>, uses a simple, polygon-expensive VRML model to represent the kelp. Collision detection or collision avoidance with several *FishWorld* entities would be very CPU-intensive for such a model. A more dynamic model that avoids these high costs is required.

c. How to Create Realistic Underwater Environment

Underwater environments dynamically alter light in fantastic ways. Surface waves create caustics that create a shimmering light show on nearly every visible object in an aquarium. Light shining through the waves creates a unique light show. Water creates reflection of surface objects. Water acts like a blue filter affecting visibility and brightness. These effects must be simulated in a realistic way.

E. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

Chapter II, Background: This chapter contains the requisite background information that supports this thesis. This information includes a description of the capabilities of and technologies used by NPSNET-V. This description is required context, because the proof of concept application executes NPSNET-V to provide network connectivity and dynamic entity discovery. Multi-Agent Simulations (MASs) and their numerous capabilities are described. Example MASs and their contributions that have provided direction to this thesis are presented. Related Monterey Bay Aquarium modeling projects are also given credit, because many of these components are reused in a proof of concept application, *FishWorld*.

Chapter III, Integrate RELATE with NPSNET-V *Model, View, and Controller*: This chapter describes the autonomous control of entities participating in NPSNET-V. This description includes a detailed study of autonomy and the many interactions of agents in the proof of concept application. Additionally, how this control is integrated into the *Model, View, and Controller* design is described. The purpose of this chapter is to assist the reader with rapidly creating new entity types with autonomous behaviors.

Chapter IV, Creating New NPSNET-V Applications : This chapter describes the major software engineering contribution of this thesis. The purpose of this chapter is

to provide the reader enough information, so that this reader can rapidly create new virtual worlds and entities for use in NPSNET-V, which can host an unlimited number of virtual worlds.

Chapter V, Proof of Concept: *FishWorld*: This chapter describes the specific contributions of the proof of concept application, *FishWorld*. Many of the features and design challenges are detailed to provide the reader ideas and solutions for the creation of new NPSNET-V virtual worlds.

Chapter VI, Conclusion: This concludes the thesis by summarizing contributions, reviewing design challenges, and providing direction for the creation of new, participating virtual environments. Additionally, this chapter describes areas that require continued effort and study. This chapter will encourage other students to participate in expanding the already formidable capabilities of NPSNET-V.

II. BACKGROUND

This chapter contains some requisite background data to create the proper context for this thesis. A description of the technologies used by NPSNET-V is provided to enhance the understanding of its capabilities. Only by understanding these technologies can an interested participant implement new virtual environments to run on NPSNET-V that fully leverage the entire range of features. After a brief definition of Multi-Agent Simulation (MAS), three example MASs are described because of their specific contributions to the proof of concept application. The final background topic is a description of various three-dimensional graphical modeling languages that can be used by NPSNET-V to render *EntityViews*. This is required because NPSNET-V is platform and graphics standard independent.

A. TECHNOLOGIES

To avoid recreating solutions to preexisting problems several technologies are used to create a proof of concept application for this thesis. MASs are an emerging technology with far-reaching applicability. It is useful for conducting experiments or simulations in situated (three-dimensional) and non-situated environments. RELATE is a Java library, developed by Kimberly Roddy and Michael Dickson at the Naval Postgraduate School, which is useful for implementing MAS in which participating agents conduct goal-based reasoning (Roddy and Dickson, 2000). A genetic algorithm is a software process for creating new agents in MAS. In order to see the results of situated MAS, a graphical representation is best. The proof of concept application uses Java3D

and VRML in order to accomplish this representation and interaction. All of these topics are discussed below.

1. Multi-Agent Simulation

Multi-Agent Simulations are containers for the interactions between autonomous agents. MAS provides the environment, situated or non-situated, and the rules that govern how the agents interact with each other and with this environment. Many MAS assign decision trees to each autonomous agent to conduct decision-making. Each agent is either assigned or chooses the best goal (the top of this data structure) to follow based on the current situation. Many MAS applications use an objective function to grade the performance of the autonomous agents. Agents can be instructed by the MAS to change their behavior if they start behaving poorly. This is the source of adaptability of agents in MASs. Due to these factors, MASs can be used to model complex environments, solve for complex algorithms, or explore emergent behavior (Ferber, 1999).

2. RELATE

RELATE is a way for an autonomous agent to implement actions in a Multi-Agent Simulation. RELATE is a Java library that contains interconnecting interfaces that, when implemented, creates a goal-based reasoning decision tree. The architect that uses RELATE can select an appropriate goal for decision determination based on weight assignments or on situational events. This architect can also have RELATE automate the selection of the active goal based on active relationships between two or more agents.

This makes RELATE a very powerful library, because it can be used to effectively model the relationships between interacting components in any simulation (Roddy and Dickson, 2000). The environment being simulated can be situational in 3-D space or non-situational.

RELATE is an autonomous-agent design-paradigm using a goal-based decision tree for action-determination (reasoning). This decision tree is a directed graph. “Active” nodes represent the path taken through the graph. These “active” nodes know about the selectable, children nodes below them. Unless directly assigned by a higher-ranking agent, parent nodes choose the best child node to create an optimal path through the decision tree.

3. Genetic Algorithms

Genetic algorithms are software processes that mimic the genetic combination of two biological, parent organisms into one or more offspring during reproduction. Alleles are segments of genetic material in biological organisms that determine a propensity for certain behaviors or physical attributes. These alleles exist in an ordered sequence comprising the organism’s genetic material. Different species of organisms may have a different sequence and numbers of alleles, but multiple instances of the same species will have the exact same number of alleles and the same sequence. The differences between like-species organisms occur in the genetic material that comprises each allele. During biological reproduction, the process generates a unique set of alleles for the offspring. This set of alleles has the exact same sequence as both of the parents, but is constructed

by a random selection of one of the two parents' alleles. The change of alleles from one parent to the other is called crossover. The number of crossovers can equal the number of alleles in a species' sequence or be fewer.

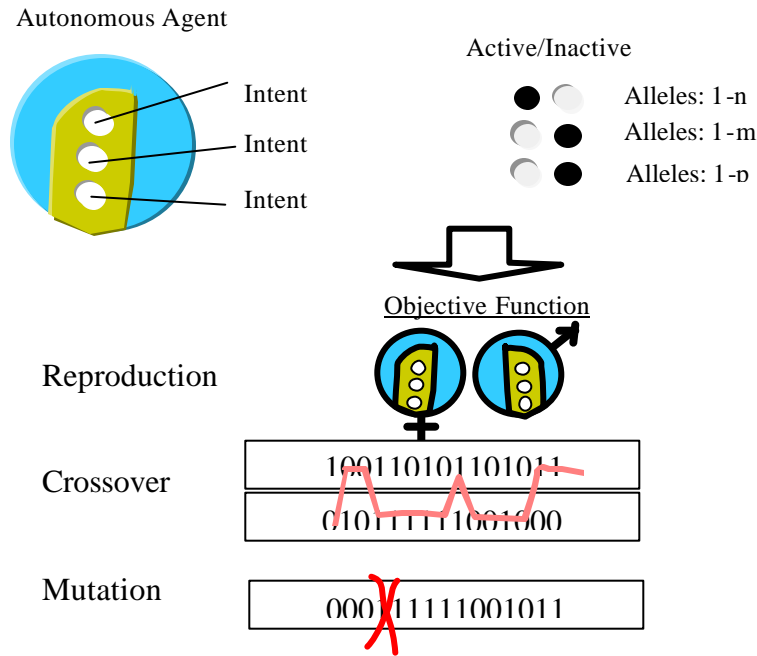


Figure 2. Genetic Algorithm.

Just as in biology, software genetic algorithms can combine the alleles of two software agents and create a new agent. The only requirements are that the agents be constructed in a way that their personalities (propensity for certain behaviors) are described by software alleles, and that the exact same allele types and sequence describe the two mating agents. Genetic algorithms select alleles from two parents and create genetic material for new offspring. The selection may undergo a random pattern of crossovers between these two parents for each allele in the sequence. The final combination can suffer genetic mutation at a rate determined by the programmer (see

Figure 2 above). Genetic algorithms can ensure survival of the fittest as in nature if the objective function used to select parents is well designed (Fisher, 1958). The introduction of mutations ensures that new personality types can be discovered (Von Neumann, 1966).

4. Java3D

Java3D™ is a Java-based library for modeling 3-D scenes. Compiled Java3D™ classes can be passed as bytes across network connections, reassembled on the other side, and executed on the receiving machine by the resident Java Virtual Machine, regardless of platform (Gosling and McGilton, 1996). Java3D™ organizes objects in the 3-D scene into a scene graph. Loaders exist that can load scene graphs constructed from a variety of graphics file formats. All of these features make Java3D™ a good choice for Networked Virtual Environments (Stapleton, 1997). These features are also what make dynamic entity loading possible. The ability to transport a graphical representation of an entity over the network, reassemble on the remote machine, and run is critical to NPSNET-V.

5. VRML

VRML stands for Virtual Reality Modeling Language. VRML is a 3D modeling language that can be viewed in web browsers with the correct installation of required plug-ins. As described above, Java3D can load scene graphs that have been constructed in other languages. VRML is one such language. With the correct loader, VRML scenes can be loaded and added into any Java3D scene graph. VRML uses a construct similar to

the scene graph used by Java3D. The VRML loader used by Java3D is essentially a text parser that reads through the VRML text file and adds the closest-corresponding Java constructs to the loading scene graph (Day, 1999). This feature allows participants of NPSNET-V virtual environments to draw graphical representations from existing repositories and libraries for the creation of new entities (Brutzman, 2000).

B. RELATED WORK

There are several completed applications and research areas that provide extensively tested solutions for areas of this thesis. For example, NPSNET-V uses technology similar to that used by the Internet to connect to a web page using a URL (Uniform Resource Locator) without prior knowledge of the page's physical location. This technology is used to implement dynamic entity discovery in NPSNET-V. The proof of concept application, *FishWorld*, relies on solutions to problems in other Multi-Agent Simulations—specifically: El Farol, Boids, and Capture The Flag.

1. NPSNET-V

NPSNET-V (Capps, 2000) is the basis for this thesis work. Its capabilities are what will be extensively explored in creating *FishWorld*. The mechanics of the NPSNET-V system are described in detail below.

a. Lightweight Directory Access Protocol (LDAP)

To ensure the ability to load and display heterogeneous entities and numerous different virtual environments, NPSNET-V uses a component-based

architecture. To provide the capability to dynamically discover and load new entities, this architecture additionally requires a system for storing and retrieving components. Because a single component storage server creates a single point of failure, replicated component storage servers are required. In the event of a server failure the components are retrieved from a replicated server to where data has been automatically copied. This also allows implementers to load-balance component storage/retrieval across several servers to support several users requesting to download a newly appearing entity simultaneously. This capability provides the requisite support to implement a large-scale virtual world.

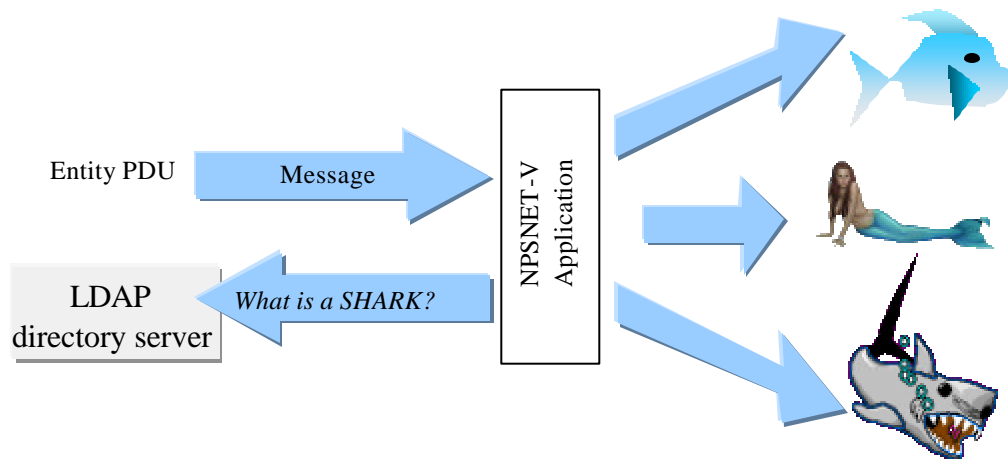


Figure 3. NPSNET-V LDAP Directory Server.

To avoid creating a system to meet these requirements, an existing, open standards solution, Lightweight Directory Access Protocol (LDAP) is used (Yeong, 1995). In addition to performing storage/retrieval services, LDAP provides a naming

service, which uses URL (Uniform Resource Locator) references to locate the stored components. Currently, the latest NPSNET-V release, which relies on a single server, does not support the replication of component storage servers. When a virtual world discovers that it must load a component it provides the LDAP server with a URL, and the LDAP server replies with the requested component data (Hodges, 1997).

NPSNET-V uses an LDAP server with the functionality described above to host entities. In order to load entities at run time, the URLs to the entities *Ghost*, *View*, and protocols are registered with the LDAP, so that receivers of packets for this entity can download and display the entity. Figure 3 above depicts the dynamic loading of entities using this service.

b. Entity Dispatcher

The NPSNET-V entity dispatcher works closely with the LDAP. When the entity dispatcher receives a new packet (see Figure 4 below), it attempts to find the entity to which the packet is addressed on the local system. If the addressee is not registered with the entity dispatcher, the entity dispatcher asks LDAP for its *Ghost* and *View* components. The URL posted on the LDAP provides the path for these objects to be serialized and transported across the net to the requesting machine. Entity dispatcher now registers the new entity and is able to pass packet information to it. Entity dispatcher notifies the application that a new entity has been registered. This exact same process is followed for new packet types as well. The LDAP works hand-in-hand with

entity dispatcher to provide dynamic entity discovery and dynamic behavior discovery (McGregor, 2001).

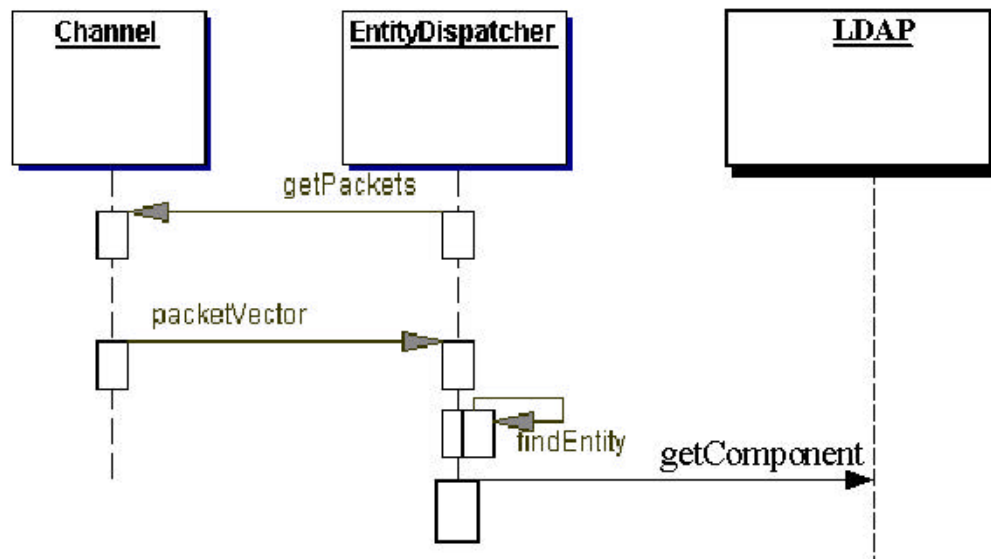


Figure 4. Entity Dispatcher Receive Sequence.

c. Area of Interest Manager

In order to meet the requirements for high scalability, every listener cannot receive every packet for all entities. NPSNET-V implements a packet filter that works dynamically by dividing the world into several areas of interest called zones. A listener in a zone only receives packets from others that are located in the same zone. When this area of interest becomes overwhelmed by network traffic, the NPSNET-V area of interest manager will divide that area into smaller areas of interest dynamically. Each area of interest will have a unique multicast channel to accomplish this. The area of interest

manager manages the distribution of multicast channels and allocating areas of interest for each entity (Wathen, 2001).

d. Dynamic Protocol/Entity Discovery

New *Entities* can be added to the system, and *Entities* are composed of *Model*, *View*, and *Controller* components. The LDAP server can store information about these new components. All the user must do is host the Java serializable code on a web server and assign the URL to the component—a subclass of *EntityGhost*, *EntityView*, or *Protocol*. When the component is registered with the LDAP server, every interested machine can download the new instance and display the *Entity* or behavior (McGregor, 2001).

e. Dynamic Network Optimization

In order to support the requirements for scalability to a large number of participating entities, NPSNET-V implements several strategies that will reduce network traffic to a manageable level. Reducing data precision can reduce packet sizes on the fly (floating-point precision data replaces double-precision data, etc.). This filtering can be turned on dynamically as network bandwidth usage increases. As already described, the world can be divided into smaller and smaller areas of interest, so that no single area of interest becomes overwhelmed by network traffic regardless of the number of entities in the overall world. The final method of managing network traffic is to implement different strategies for dead reckoning. If the network is far from full capacity, the entities may send packets to listening *Ghost* representations at frame rate. This will

provide high fidelity motion for the *Ghosts*, making their actions correspond more closely to the *Master*'s. As the network starts to become loaded, the entities can shift to a lower fidelity mode in which the *Ghost* performs dead reckoning between packets.

2. Kelp Forest Modeling Project

A 3-D model of the Monterey Bay Aquarium Kelp Forest already exists. This project modeled the tank and the fish in VRML. It is not dynamic and does not allow for dynamic interaction (Brutzman, 2001). This environment does provide many realistic models of the aquarium and fish that could be integrated into the *FishWorld* virtual world proof of concept application for NPSNET-V.

a. Static Path Animation

The motions of all the fish in this environment are pre-scripted path animations. Although a viewpoint can be changed, the behaviors of the fish never change. There is no dynamic interaction between entities, and new entity types cannot be added at run time.

b. Static Environment

The motions of the environment are also all pre-scripted. The environmental affects such as currents and collisions have no impact on the motions or the interactions on the entities. To provide a realistic virtual environment, new entities should be capable of this basic level of interaction.

3. Capture the Flag

A dynamic 3-D virtual environment populated by autonomous agents exists. The name of this application is “Capture the Flag” (Brutzman and McGregor, 2000).

“Capture the Flag” has some interesting capabilities, but it does not have the dynamic extensibilities of NPSNET-V. It lacks dynamic entity discovery. The agents that operate in the simulation are not capable of interacting with new agent types. Here are some capabilities of “Capture the Flag”.

a. Distributed Interactive Simulation

“Capture the Flag” is a networked multi- agent simulation. It incorporates a package called DIS-Java-VRML. DIS stands for Distributed Interactive Simulation. It is a standard created for military simulations and contains about twenty-seven Protocol Data Units (PDUs). The Java application simply transmits DIS Entity State PDU packets for each entity. Each entity’s view is a uniquely identified node in a VRML scene graph. The node listens for packets assigned to its entity. The node manipulates the attaching transform to correspond to the data in the packet. This is essentially a version of the *Model-View-Controller* design pattern. The use of the *Model-View-Controller* design pattern ensures appropriate data encapsulation (Brutzman and McGregor, 2000). The use of dead reckoning is a feature of DIS protocols, which allow the *Ghost* representations to continue moving in the absence of update packets. The DIS protocols are considered heavyweight, however, and carry considerable overhead as part of packets.

b. Multi-Agent Behaviors in Capture the Flag

The multi-agent behaviors in Capture The Flag are written using the RELATE Java library. The use of RELATE provides a robust interface to designing dynamic interactions between agents. The agents are arranged into two teams—red and blue. This forms the relationship at the top of the RELATE decision tree. All blue agents are able to form “blue team” relationships. Roles are assigned based on this relationship—squad leader or squad member roles. The squad leaders can assign goals such as attack or defend to squad members. The members then implement the appropriate rule to satisfy the active goal.

4. El Farol

El Farol is a multi-agent simulation design problem. It is often used as the first multi-agent programming assignment for students because it involves simple yet fascinating interactions between agents. Brian Arthur designed this problem in 1994 (Edmonds, 2001). A population of agents must decide whether to go to the El Farol bar each Thursday night. No agent likes to attend the Thursday night revelry if the bar is too crowded (i.e. more than 60% of the agents show). Therefore every agent independently predicts what the attendance will be. Agents maintain a single active predictor from a container of many. If an agent chooses, this predictor may be discarded and replaced by another. Modeling the problem to allow previewing the predictions by a preponderance of the agents would be self-defeating, because as the average of the previewed predictions falls below 60%, attendance would be high, causing the bar to be

overcrowded (Edmonds, 2001). Modeling the El Farol problem creates agents that adapt their behavior. The interactions between the adapting agents lead to the creation of dynamically emergent behavior. Both of these properties are important characteristics of the proof of concept application.

a. Adaptable Behaviors

El Farol uses an objective function to grade the participating agents. Agents that are performing poorly are instructed to change their active rules, while agents that are performing well maintain their current active rule. All El Farol agents have the exact same goal (to attend the bar); it is only the active rule (predictor) that changes, so there is essentially only one level in an El Farol agent's decision tree. When an agent upgrades to a new rule, it is the best performing rule at the time. This causes rules to continually rise and fall in performance and to continually be selected and deselected as active rule. This variance is adaptability.

b. Dynamic Emergent Behavior

As a rule increases in scoring due to improving accuracy, more agents select this rule as the current active rule. The switch to this rule by several agents causes this rule to lose effectiveness. If a majority of the agents have the same action, the main goal is violated. This cycle allows other rules that were performing poorly to start improving and increase in popularity. The behaviors of an individual agent affect interactions between the other agents participating in the simulation. This cause and effect relationship can create complex results that are difficult to predict. This is the

essence of dynamic emergent behavior. This can become even more complex if multiple goals are selectable instead of only one as in El Farol.

5. Boids

Boids is a multi-agent simulation problem of coordinating birds into a flock. The study of dynamic flocking behavior precedes the SIGGRAPH '87 Conference (Reynolds, 1987). The problem is to have the birds flock while avoiding collisions with each other and avoiding collisions with the environment. The dynamic emergent behavior that evolves from this simulation could answer several questions about bird behaviors. Birds get close enough to flock without knocking into each other. An entire flock avoids obstacles and remains a flock. Birds in a flock do not agree to a specified formation or communicate intentions. Modeling this level of complexity is the interest in implementing Boids. This is an interesting study for the implementation of *FishWorld*. The fish agents will participate in schools that are quite similar to flocks. Each involves complex interactions in three-dimensional space.

a. Flocking Behavior

This seems simple at first. If two birds see each other, then they should fly toward each other. You could alternatively have one bird mirror the other. If two birds start flying together, they may collide, though. If two birds start mirroring each other then they may simply fly around circling each other—each trying to mirror the movements of the other. Three steering behaviors have emerged in the study of flocking to coordinate individual bird's motions. In one algorithm, birds mirror the heading of the

average of the closest birds in the flock. This technique is called alignment. In another algorithm, they always want to be in the center of the flock and are, therefore, always heading for the center. This technique is called cohesion. In the final algorithm, a bird will turn to avoid crowding the closest members of his flock. This technique is called separation (Reynolds, 1999). As birds ebb and flow in the currents of the flock, collisions with each other must be avoided. This is vitally important, because these collisions could cause injuries. Some models of the flocking problem compute the result of each of these three algorithms and move the boids a distance and direction equal to the average. Other models assign an importance ranking for each algorithm and move the boid a distance and direction equal to the result of the highest ranking one.

b. Terrain Avoidance

The description of the complexity of the flocking behavior has not yet considered what occurs when an impact with an obstacle is imminent. This event could most likely kill the bird, so this goal should have a high importance value. It is possible to avoid a collision while maintaining a flock. Predictive obstacle avoidance provides the simple solution. As lead boids of a flock observe approaching obstacles, predicting avoidance provides them steering commands around the obstacle. By simply following separation, cohesion, and alignment, follow-on boids will be guided around the obstacle while maintaining a cohesive flock. Boids that are not guaranteed clearance around the obstacle will conduct predictive obstacle avoidance. In this way, the flock may split into groups to negotiate the obstacle.

C. CONCLUSION

Only by understanding these topics can the full scope of this research be realized. NPSNET-V relies on the technology provided by Java, LDAP, VRDNS, and the Internet. *FishWorld* relies on the technology of NPSNET-V, Java3DTM, and Multi-Agent Simulations to create a dynamic networked virtual environment hosting a myriad of dynamic, heterogeneous autonomous agents. An application such as *FishWorld* has the potential for far-reaching capabilities due to being constructed on such a foundation of these components. The aim of this research is to demonstrate the synergy of combining these technologies into a single application.

THIS PAGE INTENTIONALLY LEFT BLANK

III. INTEGRATE RELATE WITH NPSNET-V MODEL, VIEW, CONTROLLER

Because of the distributed nature of NPSNET-V it is vitally important to incorporate sound principles into the design of virtual worlds and entities. By using the well-proven technology of the *Model-View-Controller* (MVC) pattern for the implementation of entities in NPSNET-V, development time and testing decrease when reuse increases. The ability to load individual components for the dynamic creation of entities from the repository stored on the LDAP servers is being considered. By selecting separate MVC components from the repository of entities created with these three components, new entity types can be constructed without a single line of new code being written. For entities written with autonomous behaviors, it is critical that these behaviors be correctly incorporated into MVC pattern. Only by this technique can the correct implementation of these behaviors be available for the component creation of entities. This chapter describes one correct design.

A. RELATE IMPLEMENTATION

The proof of concept application, *FishWorld*, is created using the RELATE library. The top of the RELATE tree defines the active relationship of the agent. The Relationship node in the decision tree in RELATE actively searches to form relationships with other agents that are within sensor range. This works very well for several applications. In many of the combat models built upon RELATE, knowing which side an agent represents is very important. A friend or foe relationship forms at the top of the

decision tree for these type applications. Relationships are restricted from forming where they cause violations. This would occur if a *RedRelationship* attempted to join a *BlueRelationship* if red and blue represents opposing sides. The next node in the tree is the Role. In the applications that model leadership, roles could represent the appropriate rung in the hierarchy of the chain-of-command. A company commander role could issue platoon leader roles. Platoon leader roles would assign squad leader roles. Squad leaders would issue squad member roles. Every agent will have exactly one active role.

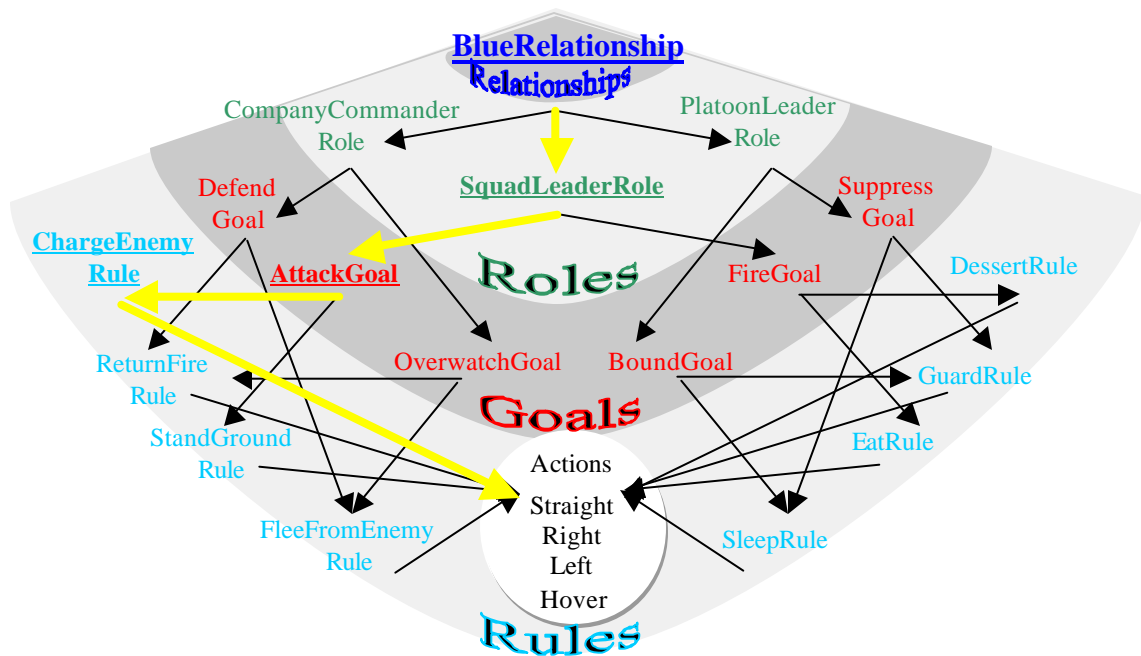


Figure 5. RELATE Decision Tree.

The next node in the tree is the Goal. The chain of command can use a goal structure to accomplish a mission. The company commander may decide to attack. He could issue this active goal to the few platoon leaders. The platoon leaders could issue

goals such as bound or provide over-watch to squad leaders. Squad leaders could issue active goals to the squad members such as attack, rush forward, or fire weapons. If active goal selection is not accomplished by assignment by a higher-ranking agent in the chain of command, then the agent's current active role simply assigns it. A container of possible goal choices are maintained in each role. Goal objects maintain a container of possible rule choices. The current active goal makes a decision which rule to use to satisfy this goal in the same way the active role decides which goal best satisfies the assigned role.

Based on the active goal, an individual agent will perform a rule that accomplishes it. The commander's active rule may be to issue commands or to move to stay in a position to control the company. A squad member may have the rule to kill an individual enemy soldier, to run to a hill, or to follow the soldier in front of him. The active Rule ends the decision process by simply picking the best action to accomplish the rule. This action may be to turn left, right, or continue straight.

B. NPSNET-V

The design of the MVC pattern provides the proper data encapsulation to easily model intelligence for NPSNET-V entities. The RELATE implementation in *FishWorld* essentially becomes the controller for the resident *FishWorld EntityMasters*. NPSNET-V *EntityMasters* have a different set of controllers than *EntityGhosts*. *EntityMasters* determine the behavior of an agent and all the representative *Ghosts*. *EntityGhosts* employ techniques that attempt to replicate a *Master's* motions and animations (*Ghost*

dead reckoning). Those algorithms that allow a *Ghost* to mirror a *Master* will be a controller on a *Ghost*. In NPSNET-V, shared state information between *EntityGhosts* and *EntityMasters* is passed through the use of protocols that are sent over the network. The controller plugs into the owning entity to set and change state information in the model by interpreting received protocols. *EntityMasters* may have a different set of state attributes (different model) than their representative *EntityGhosts*. After completing the agent implementation, the first problem was to insert this controller into an *EntityMaster*. The subsequent challenges include designing and integrating the controller for the *EntityGhost*.

1. RELATE Agent to NPSNET-V EntityMaster

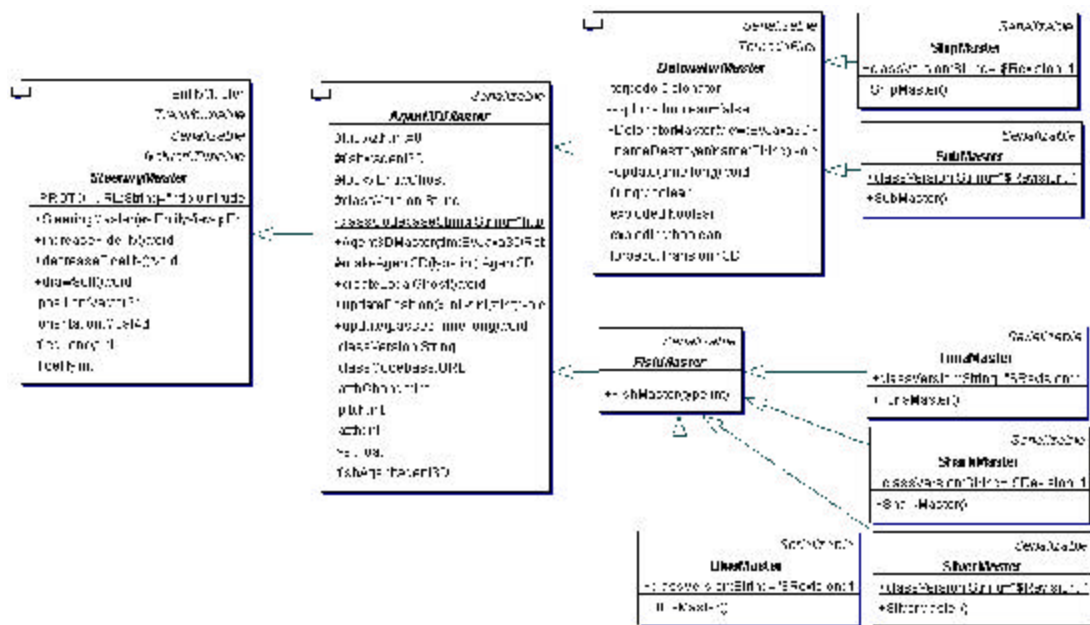


Figure 6. FishWorld EntityMasters

The design of the RELATE decision process has been described in great detail but the *EntityMaster* design must also be described. The state information that is shared between *FishWorld Masters* and *Ghosts* is described by an interface called *FishInterface*. The use of the *FishInterface* allows the resident autonomous agents to interact with any entity in the same area of interest—*Ghosts* or *Masters*. In order to collision avoid or school with other entities, spatial knowledge is required. Entities only school with others of the same type. These are two examples of the type information provided by *FishInterface* implementation.

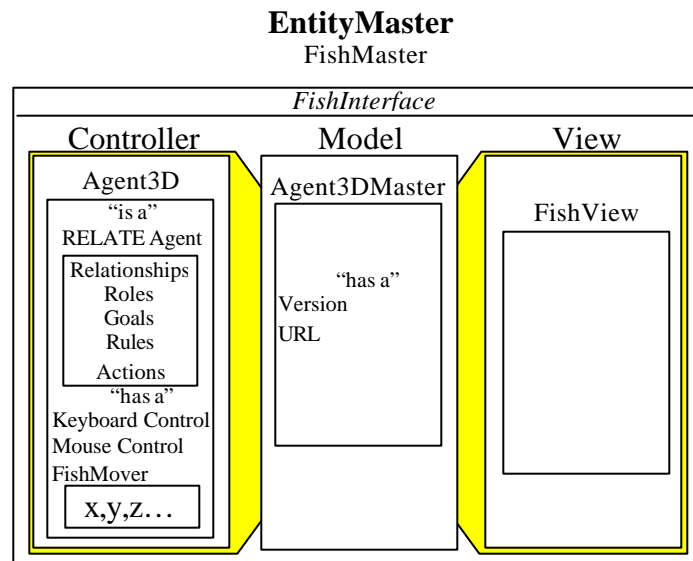


Figure 7. EntityMaster Design.

The *Master*, called *FishMaster*, forms a “has a” relationship with the RELATE Agent (controller), called an Agent3D (Figure 6 above shows this relationship). The Agent3D implements a RELATE Agent (“is a’ relationship). Additionally, because the Agent3D understands user-control and autonomous control, the Agent3D solely

encapsulates the entire controller object of the MVC design (see Figure 7 above). The *Master*, therefore, forms a “has a” relationship with the controller in *FishWorld*. The requisite world coordinate knowledge is entirely contained in an object called *FishMover*. Because the implementation of the RELATE Agent encapsulates spatial knowledge of the environment, other agents, and self, the Agent3D controller is responsible for maintaining and updating positional and orientation data for the *FishMaster*.

Logically, this spatial knowledge should be assigned to the model—not the controller. The described encapsulation was chosen, because of the tight cohesion required between the controller object and spatial knowledge. It is important to reduce overhead where possible to support the scalability requirements of NPSNET-V. The controller inquires about local coordinate position and orientation at frame rate; so function calls should be minimized. For this implementation, part of the model is essentially distributed to the controller. This description is not a violation of the *Model*, *View*, and *Controller* design, because many chunks of the model may be encapsulated into separate objects. The model could easily be distributed across many classes.

The drawback to this approach is the impact to future work on the NPSNET-V research project. A future capability is the dynamic creation of entities. Because entities are stored on the LDAP server, it is possible to introduce entities into a virtual world by using the dynamic loading capabilities. If these entities use the proper encapsulation with separate components for the *Model*, *View*, and *Controller*, mixing and matching plug-able components could create new entity types.

The *FishMaster* also “has a” view object called the *FishView*. This is described in greater detail below.

2. NPSNET-V EntityGhost

A *Ghost* is downloaded from the hosted http server from the link passed by the LDAP server to the entity dispatcher when the first packet update is received for this entity. The *Ghost* is deserialized and instantiated using the class loading capability of Java. The *Ghost* objects created for *FishWorld* implement *FishInterface*—the same interface that *Masters* implement. This will ensure that the resident *Masters* can interact with *Ghosts*—behaviors including schooling, attacking, fleeing, etc. In addition to providing a way for all agents in *FishWorld* to learn about others, *FishInterface* represents the state variables (model) that are shared between a *Master* and its representative *Ghost*. Passing this shared model from *Master* to *Ghost* every time there is a change is one way to update the *Ghost*. Passing the full *FishInterface* data structure at frame rate for more than a handful of agents would cripple the network on which it was running by consuming too much bandwidth. Simply passing the position and orientation data would have the same effect even though only position and orientation data changes at frame rate. *Ghosts* must represent their *Masters* well without heavy dependence on rapid updates and without monopolizing the CPU. There are two aspects to the *Ghost*’s controller to ensure this accuracy without the overhead of sending the model component at frame rate: autonomous control, and protocol interpretation.

It could be possible to entirely replicate the multi-agent behavior on the *Ghost* - side, but this would clearly violate the principles of scalability for large agent participation. If one hundred agents were in the same region, thrashing would likely occur between the competing threads. There could be tunable behavior for the *Ghosts*. A high fidelity autonomous behavior controller (close to the fidelity of the *Master*) could be used until threads run out of CPU. This could be the trigger to switch to a lower fidelity autonomous behavior scheme. The lowest fidelity autonomous behavior may simply provide predictions about changes. This work is not within the scope of this thesis; additional consideration is given to this requirement in Chapter VI.C, “Future Work”.

The lowest fidelity scheme and the one implemented in *FishWorld* could simply rely on steering commands and computation of dead reckoning schemes. When a *Master* turns, the new heading, velocity, and pitch angle are sent. The *Ghost* maneuvers to match these values and continues along this new vector until the next packet is received. If no steering commands are received the agent continues straight by proceeding along the last heading and velocity received. The final piece of the controller for the *Ghost* executes its own collision avoidance with the environment by asking the application for the boundaries to the local world.

The interpreting of protocols from the *Master* forms the basis for the controller component of the *Ghosts*. This interpretation results in updates to the model component. When a packet is received that contains update information for the *Ghost*, the protocol calls applicable methods. *SteeringCommand* protocol calls a steering command to update

new azimuth, pitch, and velocity state information. *FishInterface* protocol calls set methods on all the applicable state information. Having the *Ghost* snap to the correct x, y, z position of the *FishInterface* protocol would usually present unacceptable results. Instead the *Ghost* smoothly converges to the updated position. This is accomplished by taking the difference between the current and updated positions and adding a portion of this difference over the next several frames until the whole amount has been added in. This causes the *Ghost* to smoothly converge to the correct position.

The design of the *Ghost* correctly implements the MVC pattern. Part of the controller is any multi-agent behaviors that have been given to the *Ghost* entity. The autonomous behavior controller knows how to ask the application for information about the environment—questions that ensure collision avoidance with the environment. The other part is any protocol interpretation. The model component is correctly encapsulated in a separate data structure called *FishInterface*. As described above, this interface represents the shared model components between a *Master* and its *Ghosts*. The spatial and orientation data is encapsulated in a *FishMover* object located in the model component. This object was located in the *Master*'s controller component. It was possible to maintain strict adherence to the *Model*, *View*, and *Controller* design (see Figure 8 below) in this case because of the lower fidelity autonomous control of a *Ghost*. A *Ghost* would only be using autonomous control when CPU utilization supports it, so the overhead associated with the numerous function calls does not interfere with scalability.

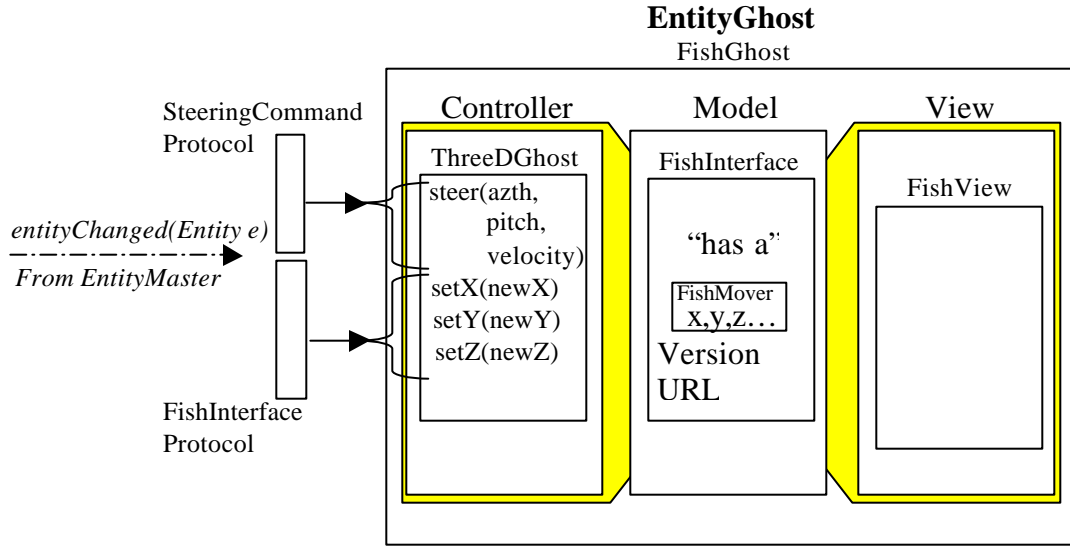


Figure 8. EntityGhost Design.

3. NPSNET-V View

The *View* object is the same for both the *Master* and *Ghost* objects. The graphics standards that have been used for the different views include Java3D and VRML scenes. These are registered with the LDAP along with the *Ghost* object and are downloaded when the *Ghost* definition is. Because the entity components are loosely coupled, simply changing the URL to a new file description changes the view for any entity.

C. CONCLUSION

This chapter successfully demonstrates one correct method for integrating RELATE's autonomous behavior into the *Model*, *View*, and *Controller* component-based design. By using this method, an entity can have coordinated autonomous or user

control. Additionally, the autonomous control component can be loaded and used by other entities participating in NPSNET-V.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CREATING NEW NPSNET-V APPLICATIONS

NPSNET-V has been constructed to provide network connectivity, dynamic entity discovery, and dynamic behavior discovery. To implement an application to fully capitalize all of the capabilities NPSNET-V provides, there are several design considerations to ponder. This application should be built using proper levels of abstraction and data encapsulation to ensure a sound design. The design should be highly cohesive and loosely coupled. Following these steps will make changes in capability easy to implement. For this reason, the application should be built in layers. Changing a layer should not affect any others.

The first layer (*AppBase* in Figure 9 below) is the layer implementing NPSNET-V. *AppBase* is a singleton with an entity dispatcher and a container holding all the entities in the application. The entity dispatcher is responsible for passing network traffic to and from participating entities.

The next layer, *AppJava3DRetained*, forms the basis for the creation of a virtual environment using Java3D. These two layers form the basis for a networked, virtual environment. The reason for this separation is a change to the network design should not affect the virtual environment. The reverse is also true. If the use of Java3D is no longer desired, the network layer should remain unchanged during a redesign—only the *AppJava3DRetained* layer would be modified to use the newly selected graphics package. *GraphicsScene* is the application layer, and contains knowledge specific to *FishWorld*.

A networked virtual environment application built using NPSNET-V needs to only concern itself with details specific to unique requirements. Network connectivity, dynamic entity discovery, and dynamic behavior discovery are handled automatically. *AppBase* notifies the newly created application of the registration and deregistration of new entities through the implementation of *EntityRegistrationListener* interface. The entity is automatically added to the scene graph.

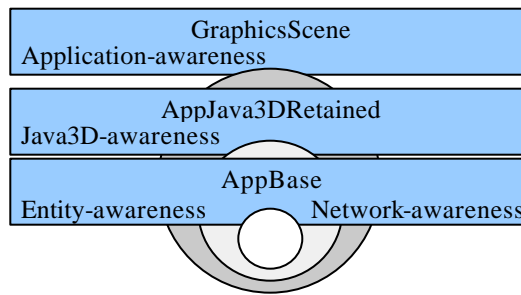


Figure 9. Virtual World Layers.

A. COLLISION DETECTION

In virtual environment applications, Entities move within the confines of the situated 3-D space of the environment. Environmental and entity-to-entity collision detection can be solved in an application-specific manner, but this approach limits the dynamic extensibility philosophy of this research. Entities would have to be changed to explore new worlds as these new applications were written for NPSNET-V using whatever unique approach decided by the creator.

1. Entity-to-Environment Collisions

To offer a reusable solution, a simple interface that all NPSNET-V applications can use, *EnvironmentInquiry*, provides robust collision detection for all uses. The “collide” method requires a 3-D world coordinate array and returns true if the passed coordinate is in collision with the world. An NPSNET-V application should simply implement the algorithm to return the correct response for the virtual environment. *AppBase* is a publicly available singleton; so any entity can easily access the running application, test to see if it implements *EnvironmentInquiry*, and perform collision detection.

2. Entity-to-Entity Collisions

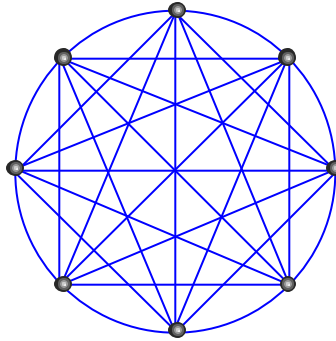


Figure 10. Entity-To-Entity Collision Detection.

Entity-to-entity collision detection is more difficult than entity-environment collision. If one hundred entities exist, and every entity asks every other entity about collision detection, ten thousand questions are asked (Figure 10). This is an order n -squared problem for n -number of entities. This scalability issue is partially solved by

NPSNET-V's area of interest manager. Areas are continually divided as processors and network bandwidths reach threshold levels. This reduces the number of entities that are known by an application. The problem is still n -squared, but n remains a manageable number.

AppBase, the singleton, can pass the container of all entities in an application to any requestor. If all entities implement the *SituatedEntity* interface, collision detection inquiries can be conducted between entities exactly as it is for *EnvironmentInquiry*. In *FishWorld*, if an entity implements *FishInterface*, even more ability is provided to assist with collision detection. *FishInterface* allows questions to be asked about orientation, position, and velocity. Entities can conduct collision avoidance more accurately if all of these questions can be asked. *FishInterface* tends to be more application specific for *FishWorld*, however, where *SituatedEntity* interface is applicable universally.

The other benefit of implementing *FishInterface* is during execution of the *FishWorld* application, each entity is handed a container of *FishInterface* objects that are within that entity's sight radius. This process occurs at frame update rate, so that an entity always knows what it can see. This entity only conducts collision avoidance with other entities in this small container—not the entire population. This creates a small efficiency gain in the *FishWorld* application. All these gains can be built into other NPSNET-V applications by copying from *FishWorld*. This would additionally make *FishWorld* entities compatible with these new worlds.

B. PHYSICS

Many applications have forces that affect the entities present in the environment. These forces may include gravity, buoyancy, air and water currents, inertia, and others. This problem has the same requirements as collision detection—offer general-purpose solutions when useful and specific solutions where required. The architect of a new virtual world application using NPSNET-V simply implements this interface replicating the physical forces affecting entities. Entities created for one world are guaranteed to be universal participants for all NPSNET-V worlds that follow this same philosophy.

For *FishWorld*, this interface is *EnvironmentInquiry*—the same one used for environmental collision detection. The method, “`applyPhysics()`” accepts a 3-D world coordinate array and component velocities and returns a corrected 3-D world coordinate array. The predominant forces in *FishWorld* include the water currents in the Monterey Bay Aquarium Kelp Forest. By calling the physics application method, entities ebb and flow with the currents of the aquarium as described in chapter II of this thesis.

C. MODEL, VIEW, CONTROLLER

This chapter describes one correct design of data encapsulation for the creation of new entity types for NPSNET-V. Understanding this will aid those interested in the creation of new entities for participation in new NPSNET-V virtual worlds. *Masters* and *Ghosts* should each be constructed from *Model*, *View*, and *Controller* components. Future work includes the dynamic creation of new entities by combining selected

downloaded components. This process would be the same as that used for dynamic entity discovery. To finish the description initiated in chapter III, this section will describe the design of the *Model*, *View*, and *Controller* components of both *Masters* and *Ghosts*.

1. Master

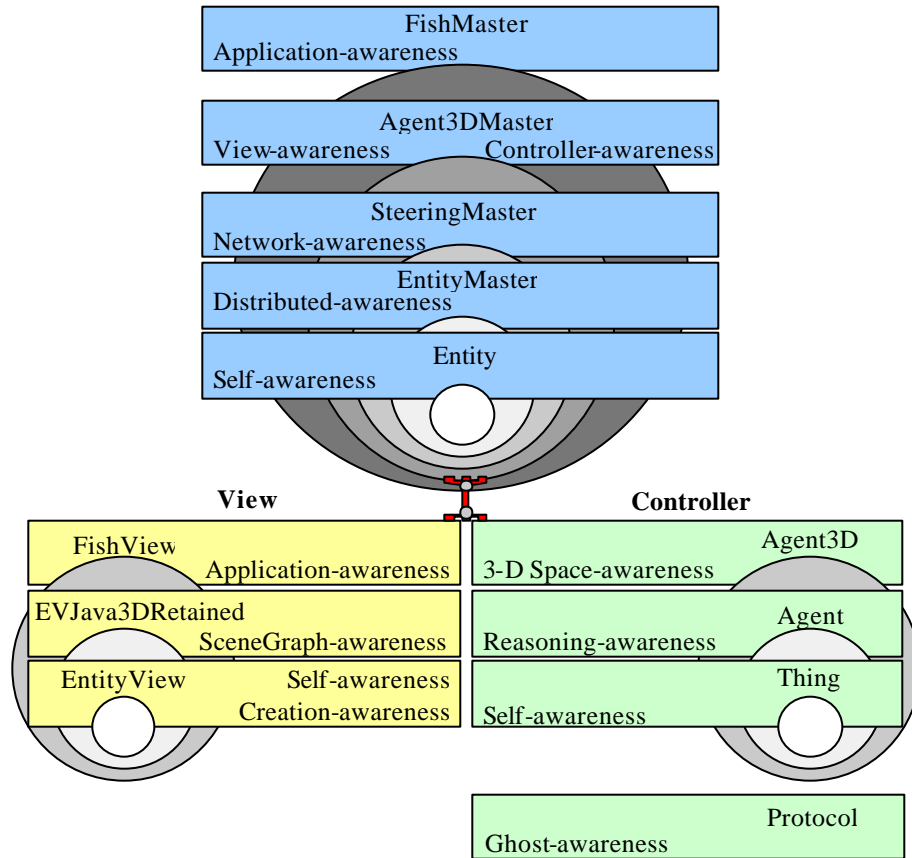


Figure 11. EntityMaster Layers.

In *FishWorld*, a *FishMaster* represents a specific fish type, but it is constructed using many layers. Similar to the layering that was described for constructing new

virtual worlds, individual entities must also be created using robust software engineering philosophies. The base class, *Entity*, in addition to containing many data structures, stores a name and a unique ID—this *Entity* must be uniquely identifiable in the entire world. This ID allows network traffic to be passed to and from *Masters* and *Ghosts*. *EntityMaster* and *EntityGhosts* extend *Entity*. At this level of abstraction, an entity either controls an unknown number of *Ghosts* as an *EntityMaster*, or is an *EntityGhost* that follows a single *EntityMaster*. Figure 11 is the referenced illustration.

At the *SteeringMaster* level, the entity understands the sending of updates to all the instances of *Ghost* representatives. The handfuls of protocols that are used by a *EntityMaster* are registered in this class, and the *SteeringMaster* knows how to convert changes to the model into a specific instance of a protocol and transmit it over the net. *Agent3Dmaster* is the culmination of the *Model*, *View*, and *Controller* for instances of *EntityMaster*. This could be broken into two different layers, but this combination follows a logical approach. The *EntityView* object is responsible for updating the scene graph based on positional and orientation data. The *Controller* manipulates this location data based on either user or autonomous input. Essentially, positional and orientation data must be shared between the *Controller* and the *View*; therefore this layer provides the correct data encapsulation. The final layer, *FishMaster*, is essentially an application layer object—it represents a specific fish type.

2. Ghost

The *EntityGhost* object has a similar structure to the *EntityMaster*, but does not require the same complexity. The *View* object for a *Ghost* is the same, but the *Controllers* and the construction of the *Ghost* model object are quite different than those for the *Master*. Figure 12 is an illustration of *EntityGhost*. *EntityGhost* extends from the base class, *Entity*. *EntityGhost* maintains an awareness of network distribution and self-creation. The *EntityGhost* class maintains knowledge that it is serialized, passed over a network connection, and reconstructed on a remote machine. It maintains state information about the status of its VRDNS registration on the LDAP server and the status of its own creation on the remote machine. The registration is required for dynamic entity discovery to operate. The *ThreeDGhost* extends from *EntityGhost* and adds the awareness of 3-D view objects, controller objects and a physics engine to move the instance of *ThreeDGhost* through 3-D situated space. These three objects are combined at this layer to provide appropriate data encapsulation. The *ThreeDGhost* is a container for this object's *View*, *Controller*, and physics subset of the *Model*—the *ThreeDGhost* contains a *View*, *Controller* and *Model* object. Changing a *ThreeDGhost*'s physics engine or *View* simply requires that this object be replaced. Crowding all these objects at this layer tend to make this a busy object, but this relationship is appropriate for the required interactions.

The appropriate *EntityView* is drawn based on the states of the model's many attributes. These states are continually modified on the *Ghost* based on the reception of

the attached protocols—the *Controllers* of the *Ghosts*. The *ThreeDGhost* provides interfaces that a controlling protocol can manipulate. The *Master* is, of course, responsible for sending out these messages to *Ghost -Controllers*, so that the states of the *Master* and corresponding *Ghosts* correlate. The *FishGhost* class extends the *ThreeDGhost* and is an application layer object; it is the specific instance of a *Ghost* fish in *FishWorld*.

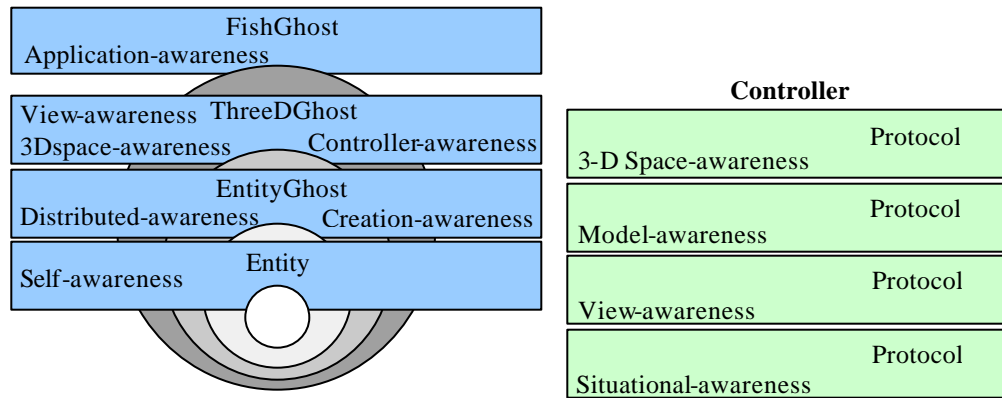


Figure 12. EntityGhost Layers.

The protocols are specific to the relationship between a *Master-Ghosts* set. The creation of new *Master* types may require the creation of new protocol types as well. This is easily handled by NPSNET-V at runtime and is one of its greatest capabilities. Protocols can include updates to a myriad of data sets including situated (position and orientation), situational (current active goal), and general (health and energy). These protocols could trigger specific animations of the view object such as firing weapons, explosions, or activating defenses. To assist with the creation of new entity types a brief

description of the protocols used in *FishWorld* is given. These four lightweight protocols accomplish all the interactions required for Fish Entities.

a. Steering Commands

Schooling *Masters* will generate several steering commands to maintain integrity of a school. Experimentation was conducted to determine the best method for sending these changes to *Ghosts*. During a turn, the change alone was being sent, but dropped packets caused the escalation of error accumulation. To rectify this, whenever a *Master* turns, the steering command contains the new heading. In the case of dropped packets, the *Ghost* can successfully turn to the correct new heading. The *Ghost* maintains angular velocity, so that the *Ghost* turns toward the new heading—it is not achieved instantaneously. When a steering command is sent, the new velocity and pitch of the *Master* is also sent. This ensures that the dead reckoning of the *Ghost* remains close to the movement of the *Master*.

The interactions between the various Fish Entities (schooling, attacking, fleeing, avoiding collisions, eating, etc.) all require situated knowledge, position and orientation data, for all participants; and a *Master* in the world is more likely to be interacting with a *Ghost* representation. Steering commands are small enough to support scalability requirements but robust enough to ensure that *Ghosts* maintain a close approximation of the *Masters'* position and orientation for accurate interaction. Fish in a school continuously make small turns to avoid collisions with other fish while maintaining the cohesion of the school. Because of this, steering commands are

generated several times a second for a schooling *Entity*, and likely be the most frequently transmitted packet in *FishWorld*.

b. Full State Protocol

Even if steering command protocols are transmitted at frame rate, error will accumulate. To rectify this, the correct situated data set must be sent periodically. Additionally, information about the current state of the *Master* model object must be updated in the *Ghost*. The full state protocol accomplishes these tasks. The *Ghost* must avoid snapping instantly to the new correct position and orientation data. The *Ghost* should converge to the true position. The sending of this full state protocol, if sent at frame rate, would provide one hundred percent data correlation between *Master* and *Ghost* (although with an unavoidable time lag). It would greatly limit the number of entities that could participate in the simulation, however. *NetworkTunable* is an interface implemented by the *SteeringMaster* that varies the rate at which the full state protocol is sent. If the processor of a participating machine or the bandwidth of a network connection reaches a threshold, the NPSNET-V area of interest manager can tune the fidelity of the simulation to provide scalability.

c. Fire Torpedo Command

To demonstrate the potential for use with military applications, submarines were created for *FishWorld*. When a *Master* submarine fires a torpedo, this state is transmitted to the *Ghost* using the fire torpedo command. This state contains information about the origin, velocity, direction, and pitch of the weapon. This allows

the *Ghost* to fully animate the torpedo employment without additional transmissions. Any collision with other agents or terrain will result in detonation. This detonation can occur on an *EntityMaster* or *EntitiyGhost*. This example requires another problem be addressed—if two entities on different machines are engaged, how is an attack on one entity by another resolved.

d. Suffer Attack

There are four possible approaches to solving the injury of one entity by another that are interacting remotely through *Ghosts*:

- Referee Server. The first method would be to use a referee system through a server. This architecture violates the principles of scalability in NPSNET-V.
- *AttackMaster-to-PreyMaster*. One *Master* could send a command to the other *Master* to suffer an attack. Due to latency and dead reckoning inaccuracy between attacking *Master* and attacking *Ghost* on the attacked *Master*'s machine, the receiving *Master* may choose to ignore the command.
- *AttackMaster -to-AttackGhosts-to- PreyMaster*. Thirdly, the attacking *Master* could notify all of its slave *Ghosts* to attack the entity prey. The *Ghost* that is resident on the prey *Master*'s machine is the one to actually carryout the attack. The attacked *Master* could then suffer an attack at the hand of the attacker. There is a slight problem, however; because the *Ghost* may be slightly out of position form the *Master* based on latency and dead reckoning inaccuracy.

- *AttackMaster -to-PreyGhost-to- PreyMaster*. Finally, this attacking *Master* could attack the local *Ghost* prey and rely on this *Ghost* to notify its owning *Master* to suffer an attack. The attacked *Ghost* could decide if the attack was valid without the penalty of inaccuracy caused by latency or dead reckoning error. The attack's validity would be measured based on the *Ghost* position—the exact same position that the attacker has pierced. If the attack was valid, the *Ghost* notifies the owning *Master* and the owning *Master* responds by changing the model. The problem here is that the prey *Master* could fight unfairly and ignore the command—of course this problem exists in all the above options.

In *FishWorld*, the final method is implemented for the attack interactions. The reasons for this include accuracy, validity checking, scalability, and ease of implementation. When the attack is completed, the *Master* registers the damage and transmits a full state protocol to notify all the *Ghost* representatives of the change.

3. View

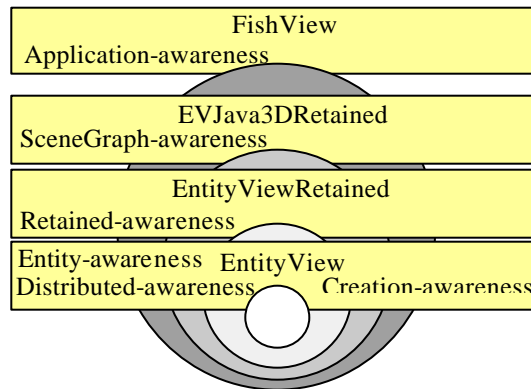


Figure 13. EntityView Layers.

Figure 13 shows that view objects in NPSNET-V all extend from *EntityView* class. *EntityView*, as a base class, has more awareness than the base class for entities. The reason for this is straightforward—all view objects are associated with a *Master* or representative *Ghosts*. Because *Masters* and their *Ghosts* use the same *View*, this *View* must be fairly robust-- knowledgeable of network registration, creation status, and awareness of controlling entity's states. To provide a generic way of updating this view object whether it be associated with a *Master* or a *Ghost*, *EntityView* implements the *EntityListener* interface. If an owning entity registers this view, whenever this entity changes the view object gets notified and is able to accurately represent the entity (this implements the observer pattern). This class must always be serializable, because it is stored on the LDAP server. It is passed as byte code, reassembled, and instantiated on every interested machine to support dynamic entity creation. The next layer represents a division into two different graphics modes—immediate or retained mode. Java3D supports the efficient retained mode, so this is the mode used in *FishWorld*. Immediate mode is used for libraries that don't provide retained mode support. *EVJava3DRetained* extends the above base classes and provides the basic structures required for insertion into a Java3D scene graph. This allows a view to translate and rotate anywhere in the virtual world to correspond to an entity's coordinate position and orientation data. At this level, multiple view types are loadable. A VRML or XML description can be parsed and loaded as a Java3D BranchGroup at runtime. In this way an entity can change appearances by simply changing the URL to the file description. The final layer of a view object is the application layer. *FishView* is essentially a specific instance of a view

object for a fish entity. This view object contains all the animations and visual operations of the owning *Master* or *Ghosts*.

4. Controller

The *Controllers* used by *Agent3Dmaster* are also constructed in layers. The reason for this is due to the complexity of adding autonomous control of an entity. RELATE was described in detail above, but it was not discussed in terms of proper data encapsulation. The base class is *Thing*, which simply possesses self-awareness. The Agent layer extends *Thing*, and has knowledge of reasoning tools supplied by RELATE. *Agent3D* is the layer that tells the Agent how and when to make a decision. An *Agent3D* object also possesses situated-space awareness. This allows the Agent to translate a decision into a physical move in the 3-D world. Because Agent objects understand user-control in addition to autonomous control, the Control object is nearly completely contained in the *Agent3D* class. The exception to this is the ability of *Ghosts* to send protocols to the *Master* to inform it of an attack or other such event. This packet may actually lead to a change in the model—in the case of an attack serious damage can be inflicted. This is why this protocol qualifies as a controller.

D. CONCLUSION

Only by utilizing sound software engineering principles can a participant take full advantage of the capabilities of NPSNET-V. Use of these principles will additionally ensure ease of incorporating future capabilities. By looking at the example components

constructed for use in *FishWorld*, new entities can be created that are capable of autonomous and user control, and new virtual worlds can be constructed to host an unlimited number of new entity types. These newly created entities can have dynamically loaded views written in any supported graphics language. Consideration must be given to entity-environment and entity-entity collisions. With only a handful of protocol types, a myriad of complex interactions are possible. By avoiding monolithic, application-specific coupling, these capabilities are extensible to dynamically loaded entities.

V. PROOF OF CONCEPT: *FISHWORLD*

This chapter describes the architecture and implementation of the FishWorld application. FishWorld was designed to demonstrate the utility of NPSNET-V's rich feature set. The previous chapters described the design of the Entities, the environment, and the mechanics for the myriad interactions. This chapter describes the design and implementation of FishWorld, including fish agent behaviors and personalities.

A. INTENT OF FISHWORLD

The intent of *FishWorld* is to construct a prototype application that fully tests the planned capabilities of NPSNET-V. Additionally, it forces the research group to complete design features and consolidate individual modules. The completed application must be robust enough to operate in concert with the advanced features of NPSNET-V while preserving the dynamics of a Multi-Agent Simulation.

1. Features

During the implementation of *FishWorld*, the features of NPSNET-V were explored and tested. The members of the NPSNET Research Group constructed these features with considerable effort. The key features that are truly noteworthy are described below.

a. Dynamic Heterogeneous Entity Discovery

New *Entity*, *View*, and protocol types that have never been seen before can be introduced to the application. The application must be robust enough to incorporate

these dynamically loaded objects. For this reason seven new entity types, seven new entity views, and four new protocols have been created:

- Shark. Entity (Rogue Predator) and View.
- Tuna. Entity (Schooling Predator) and View.
- BlueFish. Entity (Schooling Carrion Eater) and View.
- SilverFish. Entity (Schooling Carrion Eater) and View.
- Submarine. Entity (Schooling Predator) and View.
- Ship. Entity (Rogue Predator) and View.
- WeekendFishFeeder. Entity (Stationary Food Source) and View.
- SteeringCommand. Protocol.
- FishInterface. Protocol.
- FireTorpedoCommand. Protocol.
- SufferAttackCommand. Protocol.

b. Scalability

Numerous Entities may be added at runtime. NPSNET-V contains an Area of Interest Manager to allow for many networked participants. The application

must have a method for incorporating a variable number of agents at runtime. The network packets that are transmitted to relay model state information must have tunable fidelity and transmission rates to support scalability.

c. Application Generic Implementation

The implementation of the application must remain generic in order to support these myriad Entities. In order to encourage participants to generate new entity types that can participate in a myriad of NPSNET-V virtual worlds, the application must use non-monolithic solutions to challenges such as collision detection and application of physical forces such as gravity.

2. Autonomous Entity Requirements

FishWorld should provide a realistic environment. The behaviors of the participating agents should appear natural and be adaptable. Fish in the Monterey Bay Aquarium don't prescript or contrive their actions. In *FishWorld*, individual fish should not use deterministic methods to create flocking behaviors. Leader fish are not required to assign a place in a school formation to each participating fish agent. Leader fish will not decide when a school should flee from a predator. The world should not be limited to a division of only two factions—friend or foe. Even though an agent reports that he is not a predator, his actions may reveal contradictory intentions. If two predators encounter each other, one is likely to dominate—but which one? When a fish leaves a school to look for food, it should know where to start looking to rejoin. When a source of food is found, this should be a great starting location for a food search the next time

hunger is experienced. Fish generally only school with others of the same species. Fish will not necessarily flee from every other species; instead, they only flee from those that pose a threat. If these mechanisms fail, the species representing an entity type should be able to adapt to ensure survival of that species. All of these things must exist in *FishWorld*.

The remainder of this chapter describes some of the problems and design challenges that were faced in order to meet these requirements.

B. INTERACTING WITH A DYNAMIC WORLD

This section answers the question of how to create an agent that is adaptable enough to interact with new or changing environments and other agents. This set includes new agent types that may be encountered thanks to dynamic entity discovery. Agents that desire to interact in *FishWorld* must implement *FishInterface* (discussed in Chapter III).

1. Learning

To ensure survival and to give an agent the benefit of its experiences, the agents are given the ability to learn about predators in *FishWorld*. No hierarchy for predators has been pre-scripted—this is decided by who puts teeth on whom first. Not all agents flee from predators, and those that do flee do not necessarily do so unless a predator announces himself as one. In nature it is not uncommon for a creature to have an appearance that masks its true intentions. It would not be completely unexpected for

deceptive agents to be introduced into *FishWorld*. An agent should only be deceived up until the first attack, but after this occurrence, experience should teach this victim agent a thing or two. For this reason, each agent created for *FishWorld* has been given a String array it can use to store names of agent types that have attacked. This knowledge is not shared across a species, and it is not currently passed during reproduction. This creates interesting interactions (emergent behavior) between two different predator types.

Assume agent-1 and agent-3 are predator-types-1, and agent-2 and agent-4 are predator-types-2. Agent-2 will forever fear predator-types-1 if attacked by agent-1. But, at the same time, agent 3 will forever fear predator-types-2 if attacked by agent-4. Currently cannibalism does not exist in *FishWorld*.

2. Memory

Agents have the ability to store two things—food and school locations. When a social fish becomes hungry, it will leave the school to search for food. When food is found, this location is stored in memory. The social fish returns to the school location—a location stored in memory before the fish departed. This location really is only a starting point for a search for the school since the school is likely to move. When hunger returns, the fish starts the search for food by heading for the last location food was found. This is information that is not shared by a species.

The difference between this description of memory and the above section's description of learning is that nothing new is revealed about the environment or other agents for an addition to memory. The agent is simply recording and updating

information it obtains about the world. For learning, a change in the agent's behavior occurs. Memory simply helps the agent carry out the same behavior.

3. Genetic Algorithm

The use of a genetic algorithm is the only other source for adaptation in *FishWorld*. This process changes agents that fail by replacing this dead agent's personality with a combination of the best two surviving agents of the same type. The control of this combination is accomplished by a technique called crossover (discussed in chapter II). The introduction of genetic mutations allows the reintroduction of discarded or unique personalities. This process creates the ability to achieve a theoretical maximum versus stagnating at a local maximum after numerous generations. The figure below demonstrates this concept.

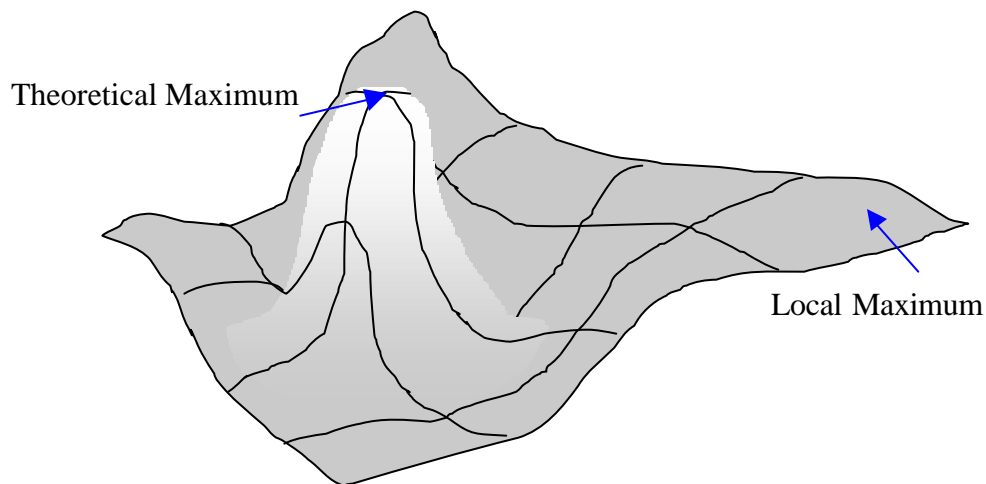


Figure 14. Local Max / Theoretical Max.

The 3D graph plots a theoretical representation of an agent's score (health) based on a combination of personality traits. After several generations of reproduction, a species may tend to become homogenized toward a personality combination that has scored well. Mutation ensures the introduction of new combinations that may result in higher net scores. Only by searching the entire graph of combinations can the theoretical maximum be found. If the environment is continuously changing, the behaviors that achieve a maximum will also vary. The use of a genetic algorithm offers a chance that the species stays competitive.

C. CREATING NEW AGENT BEHAVIORS

To meet requirements for agent interactions, unique Relationships are assigned for every unique agent type. This Relationship can then issue a Role to each agent.

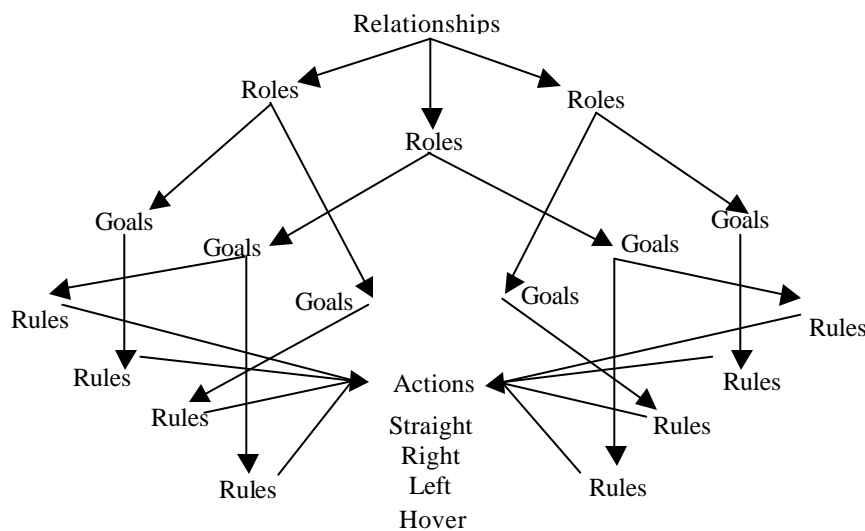


Figure 15. FishWorld Decision Tree.

1. Roles

To avoid the designation of leader fish, every agent of the same species essentially has the same Role. In many MASs, a hierarchical approach like the illustration in Figure 15 is used to simply assign agent behaviors, but for *FishWorld*, this structure should not dictate the behaviors of the individual agents. The Roles in *FishWorld*, therefore, simply become containers for an allowable goal set for a particular agent type. The agents may be predators, prey, schooling travelers, or rogue travelers. Each of these types has a particular goal set. Prey would not have an attack goal. Individualistic agents would not have a school goal. Underwater vehicles also participate in *FishWorld*. They have a torpedo goal to accomplish an attack.

The RELATE Role implementation for *FishWorld* provides a tailorable goal set for new agent types. This increases the extensibility of NPSNET-V. New users could create new agent types and introduce unique autonomous behaviors by adding or deleting new goals from their Role. For future work, users could possibly select their own goal set from a pull-down menu at run time. Being able to create new agent types with new behaviors without having to program them would extend the popularity of NPSNET-V to non-programmers.

2. Goals and Rules

Each agent has exactly one current active goal in its Role bag of many goals. Each goal has a bag of rules. Only one rule can be the current active rule (Figure 15

above or see description of RELATE in Chapter II). Active goal determination is situational. Similar to how the current situation determines the current active goal from the Role container, the active goal selects the current active rule. Goals are assigned, prioritized, and are selected based on situational metrics in accordance with their assigned priority. All agents have the same hierarchical goal set listed in order by decreasing priority in the following subparagraphs.

a. Dead Goal

This is the highest priority goal—if the conditions for death are present, this goal becomes active regardless of other factors. If an agent's health is below the threshold considered death, the active goal is the dead goal—even though no agent wishes this goal to be active. There is only one active rule possible for this goal—*floater rule*.

Floater rule is the one selected for a dead agent. The execution of this active rule of the dead goal is to sink to the bottom of the tank. When settled there, the Genetic Algorithm creates a new personality for the agent. The agent is revived and is free to swim away and to select another active goal. This is how natural selection takes place in *FishWorld*.

b. Avoid Wall Goal

Next in the level of importance is the goal to avoid collisions with the environment simply because these collisions cause tremendous injury to an agent. Additionally, the walls are not adaptable, and they do not attempt to interact with the

agents. The unmoving walls will not attempt to avoid a collision, and this is the rationale for the high level of importance for this goal. This goal becomes active if the agent is within a proximity threshold to a permanent object of the aquarium. Each agent has a different proximity threshold based on one of the factors of its personality. This can be an advantage during pursuits with a predator or a disadvantage if it leads to deadly collisions. There is only one active rule possibility for this goal--avoidance.

In the case of *FishWorld*, if a school of fish has moved up close to the edge of the aquarium, the fish closest to the obstacle would possibly be pinned. These fish would attempt to avoid the wall, putting them closer to the school. The next goal would be to avoid collision with the others. This would cause them turn back toward the wall. For these reasons, if an agent has another agent with the active goal of *AvoidWall* in its bounding sphere of radius the size of its vision, this agent will also turn away from the obstacle. This ensures that the agents remain in a cohesive school.

AvoidWall rule guides agents away from wall collisions. All obstacles in the aquarium are anchored to the bottom of the tank. Due to this, agents can pitch up and climb over all obstacles in the environment to proceed. The other requirement is to turn an agent away from the perimeter wall if this is the source of the collision. From the below figure, the aquarium walls encircle the entire environment. The challenge is to pick which way the agent should turn. To support the requirements of NPSNET-V, the solving of the collisions detection must be robust but not monopolize the CPU. The solution will be described by using the below figure of the aquarium. The aquarium is

divided in half from the center of the front cutout to the back wall. During a wall collision, the agent simply turns toward the center of the half of the aquarium it occupies.

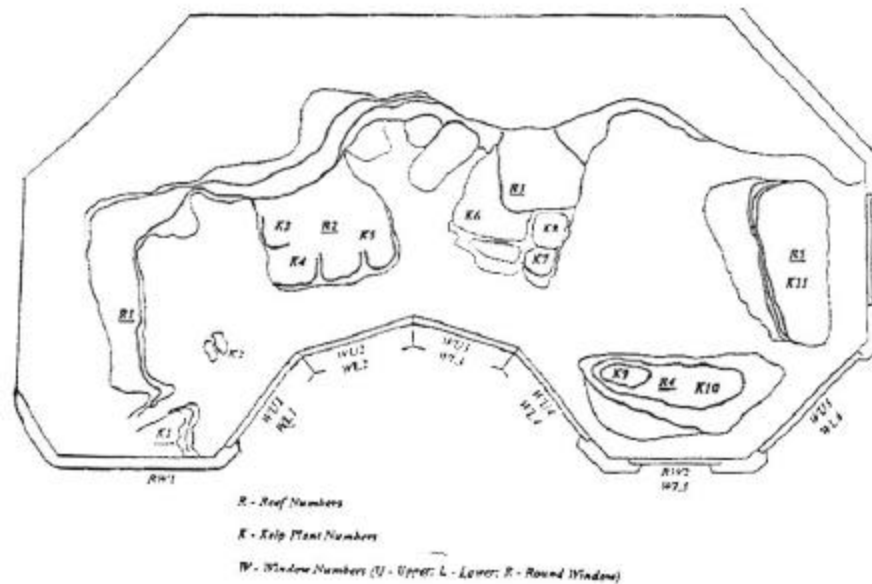


Figure 16. Aquarium Layout From Ref. [Brutzman, 2001].

c. Keyboard Control Goal

User control of the agent holds the next rung of the goal-hierarchy ladder.

The reason this rule follows the death goal, is the agent should still be susceptible to injury from attack or collisions. The reason this goal follows the wall-collision-avoid goal is because the agent should automatically return to the field of play. The lack of user skill in controlling the agent should not reduce its score for inclusion in reproduction. This effectively keeps the agent from flying through obstacles and keeps the agent from getting boxed or cornered in. The reason this goal precedes so many others including the goal for collision avoidance with other agents is that under user control, the operator is responsible for the actions of the agent. This is going to force

agents that are in close proximity to the user-controlled one to be solely responsible for collision avoidance. The user clicking on the interested agent activates this goal. There is only one possible rule—*KeyboardRule*.

The *KeyboardRule* guides the fish actions while under user control. This is accomplished through use of a Keyboard controller class. The arrow keys are used to pitch up or down and to yaw left or right. The m key is used to accelerate and the n key is used to decelerate. Pressing the escape key inactivates this goal. Additionally, pressing the space bar key while an agent is selected will toggle the agent's selection.

d. Avoid Collision Goal

The next priority is the goal to avoid collisions with other agents. Collisions with other agents inflict some damage. Many of the other goals actually attempt to bring agents together in close proximity. Due to these factors, avoiding collisions with other agents takes the next step. It has a higher priority than all the other goals that result in crowding. The factor that activates this goal is the agent's proximity to the nearest other agent. The proximity threshold for this activation is unique to each agent and is decided by one of the factors of the agent's personality. There is only one active rule for this goal—*AvoidFishRule*.

AvoidFishRule is a simple rule that ensures collision avoidance between agents. It uses pitch control to send the affected agent higher if already slightly higher than or, otherwise, lower than the agent to avoid. It uses acceleration to speed ahead if

already slightly ahead or, otherwise, to slow down. It uses directional control to turn away.

e. Flee Goal

This next goal is required purely for survival. *FleeGoal* is activated when the agent simply sees another agent that it considers a threat. This agent could consider another a threat if it is a predator and our agent flees from predators or if the same type agent has attacked it before in the past. An attack on an agent causes serious injury. Only a few attacks will completely kill an agent. This goal overrides all lower priority goals. Schooling provides protection for agents, but fleeing could cause an agent to leave the protection of the school. Because of this, a trait in the agent's personality that gives it a propensity for fleeing from predators was created. Natural selection can now decide the best technique for survival. This goal only has one rule—*FleeRule*.

FleeRule is simply an aggressive form of the rule to avoid collisions with other agents. It turns the fleeing agent away from the nearest predator.

f. Eat Goal

EatGoal has the next priority, but will not be automatically implemented. The agent must meet the threshold hunger level in order to consider eating. This hunger level is unique to each agent and is a factor in the agent's personality. Due to its priority, *EatGoal* could cause an agent to leave the protection of a school which could put the agent in danger. Eating inactivates this goal by reducing the hunger level. *EatGoal* has several rules only one of which will be active at a time:

- Circulate. The first rule involved in the pursuit of eating is circulating. This rule is used if no possible food sources are visible. The agent performs random pitch, yaw, and velocity changes until food is spotted. The frequency of directional changes is based on a factor in the agent's personality.
- Collide. *CollideRule* is used if the agent is pursuing dinner, but the pursued agent is in the protection of a school at the time of the planned attack. An attacker is unable to focus on one member of a complex school. This rule is the same as the avoid fish rule described above.
- Attack. If no dead agents are in sight and we see prey, the active rule is attack. This rule either takes the agent to the nearest prospect or to the weakest prey seen. This propensity is decided by the factors of the agent's personality. This rule steers pitch and yaw until the agent closes to within bite range of the target. Acceleration is used if the bait is in front, and deceleration is used if it is behind the agent. When bite range is achieved, the agent attacks the prey causing injury until the agent dies.
- Join Food. *JoinFood* rule guides the agent into very close proximity to a dead agent—close enough to take a bite. This rule can only be called if the acting agent sees a dead one. This agent could have been killed by any means—killed by another attacker, death by injury, killed by an attack.
- Eat. *EatRule* attempts to obtain a meal for the agent in consideration. The *EatRule* can only be called if a dead agent is in bite range. This is the rule that brings

satisfaction to the eat goal. Eating satisfies this goal by reducing hunger below a threshold to allow the next highest priority goal to be selected.

g. School Goal

SchoolGoal is the goal that schooling fish agents use as their modus operandi. Schooling can only occur with other agents of the same type that are not dead.

The possible rules are these:

- Follow Closest
- Follow Aggregate
- Follow Leader
- Follow Random
- Follow Flee
- Circulate
- Join
- Collide
- Crowd Control
- Wall Collide.

Follow closest, *follow aggregate*, *follow leader*, and *follow random* are rules that are selected based on the strongest schooling factor of the agent's personality. These rules can only become active if the agent is in the presence of the required number of same-type agents that are schooling, under user control, or avoiding wall collisions. This number, surprisingly, is unique for each agent type and is a factor in the agent's personality. *Follow closest*, *follow random* and *follow leader* are rules that move the follower agent to exactly mimic the other agent it is following. *Follow leader* is a mirroring of the agent that is furthest in front of our follower agent that it can see. If an agent is in front, it makes random movements. *Follow aggregate* moves to the location where the center of the school will be in one step. This is based on the average x, y, z coordinates, pitch, roll, yaw orientations, and velocity of all the participating agents.

Follow flee is a rule implemented in an attempt to keep fleeing agents together in a school. If a schooling agent is not fleeing but sees one that is, the active rule becomes *follow flee*. The follower agent attempts to swim to where the fleeing agent will be in one step.

Wall collide seems like a strange rule to include in school goal. This is to avoid having agents becoming pinned between the rest of a school and the boundaries of the aquarium. If a schooling agent sees another agent that is trying to avoid a collision with the wall, *wall collide* becomes his active rule. This is only applicable to agents that are not close enough to a wall to activate the *Avoid Wall* goal. In this way, the entire school moves gracefully away from the collision.

Crowd control is a rule used if a fish is in a deeply populated school. The number of fish required to set this threshold is uniquely determined for each agent and is a factor in the agent's personality. Each agent has a random coordinate it heads toward when the threshold is breached. This protects agents from suffering injury from collisions with other agents.

Collide rule is used if the agent is schooling, but the schooling agent must be within close proximity threshold of another. This threshold is uniquely determined for each agent and is a factor in the agent's personality. This rule is the same as the avoid fish rule described above. This rule is used in this way to reduce complexity and computational expense. Some other algorithms compute up to three vectors— one for school center, one for collision avoidance, and one to match the heading of the nearest neighbor. The resultant action of the agent is the average of these three. This method adds undo complexity ($3N^2$) and would not meet the scalability requirements for NPSNET-V.

The next rule involved in schooling is *join rule*. This rule is used if a possible schoolmate is visible and the deciding agent is not in the presence of the requisite number of others to constitute a school. The agent follows this rule until enough are in a clump large enough to constitute a school. Essentially, the agent swims toward the farthest prospect that it can see. Swimming toward the closest it sees would cause agents to continually swim around in pairs—never forming schools.

The final rule involved in the pursuit of schooling is circulating. This rule is used if no possible schoolmates are visible. The agent performs random pitch, yaw, and velocity changes until an agent to join is spotted. The frequency of directional changes is based on a factor in the agent's personality.

h. Cruise Goal

Cruise goal is the goal used by agents that do not school. They simply follow one rule—cruise rule. The agent performs random pitch, yaw, and velocity changes until the current active goal is changed. The frequency of directional changes is based on a factor in the agent's personality. This frequency was tuned for each agent type to create a realistic visual representation.

3. Actions

For the implementation of RELATE, iterating through the decision tree described above results in the creation of an object called an action. There are only a handful of actions required to create all the complex behaviors of *FishWorld*: turn left, turn right, slip left (half-left turn), slip right (half-right turn), straight, and hover. Within each action, an agent may accelerate or decelerate, and climb or descend. The agent must maintain spatial knowledge of the environment, other agents, and self. This knowledge must include the world 3-D coordinate system and its own local coordinate system, so that the agent knows its proximity to collisions, other objects, and other agents. The result of an action occurs at frame rate and results in the actual manipulation of the agents' position and orientation within the world coordinate system.

D. CREATING DISTINCT PERSONALITIES

The agents in *FishWorld* have unique personalities that create specific propensities for certain actions. For every personality trait, there are advantages and disadvantages. Instead of hand-tuning the agents to achieve specific, desirable interactions, the personalities are assigned random floating-point precision values from zero to one during agent creation. The entire set of propensities comprises an agent's behavior. The agents eventually achieve tuned behavior after many generations. Currently this information is not saved; so this process starts from scratch every time the application is run. This tuning is accomplished by natural selection.

1. Aggressive

With this trait, an agent won't be as likely to flee a predator, or follow a fleeing agent in the same school. An advantage to this trait is that an agent with a high aggressiveness may actually attack a pursuer and establish dominance. The disadvantage to this trait is that the owning agent is more likely to get eaten.

2. Collision Wall

The personality establishes how large a buffer this agent wants between self and the aquarium. The advantage to having a high value is that the owning agent won't be as likely to collide which causes damage. The disadvantage to this personality is that the agent won't be able to find protection against the edge of rocks or aquarium walls where larger predators could never pursue.

3. Collision Fish

This personality establishes the size of the buffer this agent wants between self and other agents. The advantage to this trait is that the agent won't be as likely to collide which causes damage. The disadvantage mostly affects social agents desiring to flock. Because the agent won't tolerate densely populated schools, it will scramble toward the outskirts of a school making it a possible victim of a predator.

4. Schooling Preferences

Several schooling schemes are used in *FishWorld*. No one technique is better than another, but agents in one type school will display different traveling characteristics than another. *FollowTheLeader* schooling creates a progressive traveling technique that quickly covers ground. This seems to be because agents are focused on following the agent that is out front. An agent out front in the lead simply avoids collisions with terrain. *FollowClosest* schooling moves nearly as quickly and progressively as *FollowTheLeader*. Agents concentrate on mirroring the pitch, azimuth, and velocity of the closest other agent it sees. *FollowAverage* schooling tends to mimic fish in small home aquariums. The agents constantly swim towards the center of the school of agents that are within sight. This school is chaotic and tends to stay in one general area. Each agent has a value for each of these traits—leadership, *followClosest*, and *followAverage*. The highest value decides which schooling technique is used.

5. Twistedness

This value is used when an agent is searching or cruising to decide how often the owning agent will turn either left or right versus heading straight. High twistedness causes the agent to conduct more thorough searches while lower values causes the agent to search over larger areas in the same amount of time.

6. Hungriness

This value determines how often the owning agent will feed. This could cause an agent to leave the protection of a school more often in order to feed. The advantage for this behavior is that the agent has more energy and is able to swim faster. This value is separate from the actual hunger value that contributes to active goal selection.

7. Blindness

Blindness determines how far an agent can see. An agent that has a larger sight radius is more skittish to the point of appearing neurotic. It sees predators farther away. It schools based on a larger number of agents. It can see food sources at greater distances. Blind agents tend to be oblivious to numerous changes that occur around them. This gives apparent purpose and stability to their actions.

E. CONCLUSION

The design used to create *FishWorld* ensures that the environment contains continuously adapting agents. By creating autonomous agents that can learn, have

memory, have distinct personalities, and can reproduce using a genetic algorithm, an unlimited number of entity variations are possible. These variations between entities change the interactions that occur in the VE, and these myriad interactions develop into emergent behaviors that demonstrate a complexity far more advanced than any of the algorithms used to produce the individual behaviors. Avoiding application-specific programming techniques, newly created entities can participate in emerging NPSNET-V virtual worlds and maintain the myriad interactions.

VI. CONCLUSION

This chapter reviews the accomplishments of this research and highlights areas that require more study. Additionally, this chapter reports the status of the thesis statement as the culmination of this effort.

A. RESULTS

The result of this thesis is *FishWorld*—the proof of concept application. The application includes realistic fish agents with realistic schooling, attacking, fleeing, and searching behaviors. Entities successfully respond to both user and autonomous control, and autonomous agent controllers correctly interact with agents that are under user-control. Agents correctly retain memory and learning events. The *Ghost* representations on remote machines closely mimic the actions of the *Master*. When *Masters* operating on separate machines engage in battle across the network, the resulting damage is coordinated accurately from *Ghost* attacked on the remote machine to the owning *Master*. The representation of underwater vehicles launching torpedoes is accurate on remote machines. *Masters* interact with *Ghosts* to form schools or engage in attacks.

In the area of component loading, dynamic entity and protocol discovery works correctly. Heterogeneous entities enter *FishWorld* without complaint. Entity registration and deregistration with the entity dispatcher operates as entities arrive and depart an application's area of interest. View objects remove themselves from the scene graph as

entities are deregistered from the application. VRML objects are correctly rendered in the scene graph.

In the area of network performance, Area of Interest Management correctly divides the world, and this distribution works correctly across the participating networked computers. Variable packet fidelity tuning operates correctly to reduce network bandwidth requirements. Variable packet frequency tuning successfully responds to network status commands to further reduce network throughput requirements. Scalability performance still needs to be tested, but by running several *Masters* on each of three participating machines on the same LAN, CPU performance faltered before network performance. This tends to indicate that NPSNET-V can achieve the goal of scalability levels that are unprecedented (Wathen, 2001).

B. CONCLUSION

All of the capabilities of NPSNET-V and the RELATE MAS were successfully incorporated together into *FishWorld*. There is not one capability that failed, though there are many that have not been fully tested. The areas that require further study, experimentation, or development are listed below in the “Future Work” section. The conclusion of this thesis, therefore, is that by combining a fully dynamic, scalable networked virtual environment (VE) with an interactive multi-agent simulation architecture, it is possible to develop virtual environments supporting a large number of dynamic, heterogeneous entities with complex, adaptable, and interactive behaviors.

C. FUTURE WORK

There are several areas requiring more work that could greatly enhance or empirically measure the capabilities of NPSNET-V. Several experiments could be conducted to determine exactly what the scalability limits of NPSNET-V are.

1. Agent Ghost Controller

Integrating agent code into the Ghost controller component would reduce the requirement for packet transmission and packet latency. The *Ghost* currently only has basic dead reckoning behaviors. If the CPU running these *Ghosts* can support this additional code for each one, running the same agent code for the *Master* and *Ghost* might reduce the need for network packet transmission while providing accurate, or at least plausible, representation on the participating machines. This might also effectively reduce latency issues between a *Master* and *Ghosts*. Unfortunately an agent's behavior in *FishWorld* is not deterministic—it is situational and probabilistic. Lag causes a *Ghost* and *Master*'s perceived environments to be slightly different resulting in their actions diverging. For this reason, there remains a requirement for some network traffic. Different levels of fidelity of agent code could be assimilated into the *Ghost*. The network status thread could then adjust each *EntityGhost*'s agent behavior up or down to tune the network throughput to balance sufficient fidelity requirements against scalability requirements for the number of participants.

2. Dead Reckoning

An experiment with and an analysis of various Ghost dead-reckoning algorithms would determine their accuracy. The current algorithm implemented for *Ghost* agents in *FishWorld* simply receives a *SteeringCommand* from the *Master*. The *Ghost* steers to match the passed heading, accelerates to the passed velocity, and noses over to match the passed pitch. The *Ghost* continues along this vector until the next steering command is sent. A position update is sent at a frequency controlled by the network status thread. When the *Ghost* receives this update, it converges to the passed correct location. This method creates smooth movements for the *Ghost* that appears very accurate when compared to the *Master* on side-by-side monitors, but formal testing is required to determine the accuracy. New techniques should be introduced and compared for accuracy and network bandwidth requirements. CPU usage should also be evaluated.

3. Agent Network Tuning

Generating an agent to control network tuning could provide an optimal speed/quality balance. Currently there are three variables that can be changed to reduce network throughput requirements: packet fidelity, packet frequency, and Area of Interest zone management. Changes in one area affect the others and affect simulation fidelity. Thresholds must be determined to complete effective tuning with the smallest changes that maximize simulation accuracy. When network throughput reaches some threshold, expanding the zones in an area of interest management scheme will essentially divide the traffic that was present on one channel by eight. This is the single most dramatic change

that can be accomplished, but this reduces the number of agents that can be seen to the number present in the same geometric region. It is likely that this change will provide enough bandwidth to allow throttling up of fidelity and packet frequency. The results of varying packet fidelity and packet frequency depend on the sizes of the specific packets being sent, but for large packet sizes, the savings available is substantial. The job of the network-control agent would be to optimize the tuning of the various techniques to provide the most realism while being the least intrusive.

4. Scalability Study

Scalability comparisons between NPSNET-IV and NPSNET-V for similar applications should be conducted to compare the network loads on each. These comparisons may be the most accurate way to estimate NPSNET-V's scalability limits.

5. Code-less Agent Creation

Creating an agent factory for autonomous agent behavior would result in code-less agent creation. Currently, the main differences between sharks, tuna, blue, and silver fish are variations in RELATE Roles, Goals, and Rules. Sharks are rogue predators, so they do not have *SchoolGoals* available as part of their Role. Blue and silver fish are schooling carrion eaters. They do not have *AttackGoals* available as part of their Role. Tuna are schooling predators, so they have all of these goals available. The creation of an agent factory could allow users to tailor behaviors for their entity. To control active goal selection, Goals could be assigned a simple priority rating like an Interrupt Request

process for an operating system. Behaviors could be continually added to a repository to allow for the creation of highly complex behaviors and interactions. This agent factory could also allow for the tailoring of personality traits during agent creation. Tailoring a personality can create quite divergent behavior even among same-type agents.

6. Component Loading

Designing and implementing a component loading technique would ease the process for entity-creation and world-discovery. If every entity were created using the same *Model, View, Controller* component design pattern, then new entity types could be constructed by combining components from the repository that already exists on the LDAP server. Work in this area could allow non-programmers to create networked agents with tailored behaviors and view objects.

7. Security

A distributed security system should be designed and implemented to protect against malicious agents in NPSNET-V. Because agent code is serialized into bytes, distributed over a network connection, reconstructed, and instantiated on the other side, it is clearly possible for malicious code to be run on participating machines. JavaTM provides some basic security against many types of attacks through the Java Virtual Machine (JVM), but this security falls short of absolute protection. A robust protection capability would ensure that users operate NPSNET-V securely and safely. The current system trusts all loaded code.

8. New Virtual Worlds

NPSNET-V can host new virtual environment applications. NPSNET-V provides the capability to host several virtual worlds simultaneously to include the capability to move between worlds. NPSNET-V is essentially a laboratory for research in modeling and simulation. Some examples of its usefulness include these:

a. Warfighting Experiment (WE)

Current WE systems are quite monolithic and require extensive modification to host new weapon systems and their myriad interactions. These modifications require exhaustive testing and evaluation, and present the possibility of introducing or uncovering bugs. The iteration through the battery of test required for a WE requires human cognition and interaction. NPSNET-V is not crippled by these limitations. The new weapon system being tested could be given selectable autonomous or user-control. The battlefield could be evaluated to determine the impact of the new system, and to measure how opposing systems adapt to this newly introduced capability. The use of genetic algorithms, learning, and memory could provide the adaptability to guide the course of the tests without requiring or limiting the conduct of the experiment to human input. The result of this type experiment would provide baseline values for weapon effectiveness, standoff ranges, system's range and required fuel capacity, armor protection, radar and IR stealth, speed, maneuverability, size, etc. These values could correspond to an agent's personality traits. The conduct of the experiment would essentially be tests of various personality sets. Because of dynamic entity discovery, the

conduct of the experiment could use heterogeneous entities distributed across the network from laboratories in various locations.

b. Experiment with Tactics

All U.S. military services conduct experiments with tactics for situations over the entire spectrum of combat operations. This includes the current trend toward peacekeeping operations. Similar to the conduct of Warfighting Experiments, the adaptability of agents could guide the course of tactics experiments. The difference in the conduct of these experiments is not with varying the personality traits of the agents, but with varying the Role, Goals, Rules, and therefore, the resultant action-selection of the agent. The best behavior set and the conditions used for this action determination will result in the highest score and provide a baseline set of tactics.

The trick is to provide a decoupled behavior set and action-determination system. The behavior set could be compiled from a repository of Roles, Goals, Rules, and Actions. The determination system could be compiled from a set of trigger events and metrics. Extensible Markup Language (XML) could provide the solution for implementation of such a determination system, and could therefore be loaded and modified at runtime. This would accurately correlate to the use of Rules of Engagement (ROE) in current NATO military operations. The union of these triggers to the appropriate behaviors could be initially constructed by human intervention, but through the adaptability of agents, could evolve over several iterations to reveal innovative

solutions. As hostilities escalate, agents could trade-up to new behavior sets to create realistic complexity.

c. Experiment with Systems

Logistic, command and control, and communication systems share certain characteristics that make them similar. Each has flow from one location to another using a distribution system involving several nodes that initiate, control, and monitor this flow. The architecture of the system of nodes may be hierarchical or randomly constructed. Objects that are passed along paths between nodes would have differing priorities, and therefore, be handled differently. The reason NPSNET-V is the best choice to conduct these type experiments is it can allow a process to be visualized, and with dynamic entity discovery, new nodes and new node types can be dynamically loaded at runtime. Autonomous control can be used to simulate the operation of nodes, and the use of adaptation could introduce efficiency gains into the process. Simulation of damaged or destroyed nodes or pathways could be conducted to discover new or test current contingencies. The results of these experiments could produce benefits on par with those from the war-fighting or tactics experiments.

GLOSSARY

- Agent - A software object that perceives its environment through sensors and acts upon that environment through effectors to achieve one or more goals.
- Model – A description or analogy used to help visualize something that cannot be directly observed.
- Coordination – The act of managing interdependencies between activities performed to achieve a goal.
- Simulation – A method for implementing a model to play out the represented behavior over time.
- Adaptation – The process of modifying one's behavior over time to advantageously form a better fit to the environment.
- Complex adaptive system (CAS) – A self-organizing system that maintains coherence in a changing environment through interactions and adaptation.
- Learning - The acquisition of knowledge, formation of associations, and modification of behavior to improve performance based on exposure to and exploration of the environment.
- Evolution – A process of continuous change from a lower, simpler, or worse state to a higher, more complex, or better state.
- Multi-agent system (MAS) – A system in which several interacting, intelligent agents pursue some set of goals or perform some set of tasks.
- MAS simulation – A rich, bottom-up modeling technique that uses diverse, multiple agents to imitate selected aspects of the real world system's active components.
- Relationship – The assembly of relations, i.e. understandings and/or commitments, between mutually interested parties that link certain individuals to others.

LIST OF REFERENCES

- Brutzman, D. (2000). *The Virtual Reality Modeling Language and Java*.
<http://www.web3D.org/WorkingGroups/vrtp/docs/vrmljava.pdf> (23 July 2001).
- Brutzman, D. (2001). *Kelp Forest Exhibit Modeling Project*.
<http://web.nps.navy.mil/~brutzman/kelp/> (20 July 2001).
- Brutzman, D. and McGregor, D. (2000). *Distributed Interactive Simulation DIS-Java-VRML Working Group*. <http://www.web3d.org/WorkingGroups/vrtp/dis-java-vrml/> (23 July 2001).
- Buschman, Frank et al (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, New York, NY: John Wiley & Son Ltd.
- Capps, M., et al. (2000). *NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments*. IEEE Computer Graphics and Applications, Volume: 20 Issue: 5, Sept.- Oct. 2000.
- Day, B. (1999). *3D Graphics Programming in Java: Part 2, Advanced Java 3D*.
http://www.javaworld.com/javaworld/jw-01-1999/jw-01-media_p.html (23 July 2001).
- Edmonds, B. (2001). *Gossip, Sexual Recombination and the El Farol Bar: Modeling the Emergence of Heterogeneity*.
http://www.cpm.mmu.ac.uk/~bruce/emhet/emhet_1.html (19 Jul 2001).
- Ferber, J. (1999). *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*, (English edition) Harlow, England: Addison-Wesley.
- Fisher, R. (1958). *The Genetical Theory of Natural Selection*. New York, NY: Dover Publications.
- Gosling, J., and McGilton, H. (1996). *The Java Language Environment*.
<http://java.sun.com/docs/white/langenv/index.html> (23 July 2001).
- Hiles, J. (2000). *Course Notes for MV-4015 Agent-Based Autonomous Behavior for Simulations*. Winter, 2000, Naval Postgraduate School.

- Hodges, J. (1997). *Introduction to Directories and the Lightweight Directory Access Protocol*. <http://www.stanford.edu/~hodges/talks/mactivity.ldap.97/index2.html> (23 July 2001).
- Macedonia, M. et al. (1995). *NPSNET: A Network Software Architecture for Large-Scale Virtual Environments*. Presence, Vol. 3.
- McGregor, D. (2001). *NPSNET-V*. <http://www.npsnet.org/~npsnet/v/index.html> (23 July 2001).
- Reynolds, C. (1987). *Flocks, Herds, and Schools: A Distributed Behavioral Model*. <http://www.red3d.com/cwr/papers/1987/SIGGRAPH87.pdf> (23 July 01).
- Reynolds, C. (1999). *Boids: Background and Update*. <http://www.red3d.com/cwr/boids/> (30 Jul 00).
- Roddy, K. and Dickson, M. (2000) Modeling Human And Organizational Behavior Using a Relation-Centric Multi-Agent System Design Paradigm. Masters Thesis, Naval Post Graduate School, CA.
- Singhal, S. and Zyda, M. (1999). *Networked Virtual Environments*, Siggraph Series. ACM Press Books.
- Stapleton, Lisa (1997). *If a Tree Falls in the New Java™ Media API*. <http://developer.java.sun.com/developer/technicalArticles/Media/3DGraphicsAPI/> (23 July 2001).
- Von Neumann, J. (1966). *Theory of Self-Reproducing Automata*. Edited and completed by Arthur Burks. Urbana, IL: University of Illinois Press.
- Wathen, M. (2001) Dynamic Scalable Network Area Of Interest Manager for Virtual Worlds. Masters Thesis, Naval Postgraduate School, CA.
- Yeong, W., et al. (1995). *Lightweight Directory Access Protocol*. ISODE Consortium, March 1995.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, VA
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Michael Zyda
Modeling, Virtual Environments & Simulation (MOVES) Academic Group
Naval Postgraduate School
Monterey, California
4. Michael Capps
Computer Science
Naval Postgraduate School
Monterey, California
5. Don McGregor
Modeling, Virtual Environments & Simulation (MOVES) Academic Group
Naval Postgraduate School
Monterey, California
6. Don Brutzman
Undersea Science and Technology Academic Group
Naval Postgraduate School
Monterey, California
7. Mr. John Hiles
MOVES Academic Group
Naval Postgraduate School
Monterey, California
8. MAJ David B. Washington
Naval Postgraduate School
Monterey, California